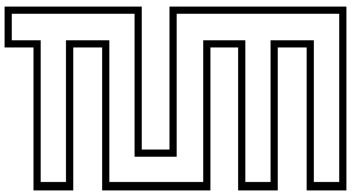


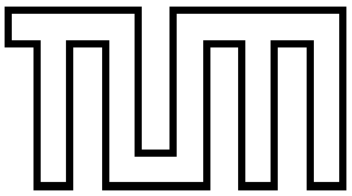
Technische Universität München

Fakultät für Informatik

Diplomarbeit in Informatik

Vergleich zwischen konventionellen und formalen Methoden zur Software-Qualitätssicherung an einem Beispiel aus dem Bereich Automotive

Daniel Reiß



Technische Universität München

Fakultät für Informatik

Diplomarbeit in Informatik

Vergleich zwischen konventionellen und formalen Methoden zur Software-Qualitätssicherung an einem Beispiel aus dem Bereich Automotive

Diplomand:	Daniel Reiß
Aufgabensteller:	Prof. Dr. Manfred Broy
Betreuer:	Dr. Jan Jürjens
	David Trachtenherz
Abgabe:	15. Mai 2006

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

(Ort, Datum)	(Unterschrift)

Zusammenfassung

Der wachsende Anteil von Software in eingebetteten Systemen prägt die heutige Automobilindustrie. Dort erfordern sicherheitskritische Software-Anwendungen ein hohes Mass an Zuverlässigkeit. Um dieses Ziel zu erreichen, erfolgt ein intensiver Einsatz von Testverfahren.

Im Rahmen dieser Arbeit werden Software-Qualitätssicherungsmethoden bewertet. Als Fallstudie dient die Softwarekomponente eines Türsteuergerätes, auf die verschiedene Verfahren angewandt werden. Inbesondere erfolgt die Evaluierung von Modelchecking als Verifikationsverfahren akademischen Ursprungs im Vergleich zu konventionellen Qualitätssicherungsmethoden wie Simulation und Whitebox-Tests mit Überdeckungsgradanalyse. Die dabei gewonnenen Erkenntnisse werden gegenübergestellt, um Aussagen über den zukünftigen Einsatz der jeweiligen Verfahren treffen zu können. Die Ergebnisse lassen sich auch auf die Qualitätssicherung von Software für technische Systeme abseits der Automobilindustrie übertragen.

Inhaltsverzeichnis

Inhaltsverzeichnis **5**

1 Einleitung **8**
1.1 Aufgabenstellung 9
1.2 Struktur der Diplomarbeit 9

2 Aspekte der Software Entwicklung **11**
2.1 Grundlegende Begriffe 11
2.2 Modellbasierte Entwicklung 14
2.3 Software-Entwicklung im Bereich Automotive 16

3 Konventionelle Qualitätssicherungsverfahren **22**
3.1 Qualitätssicherungsrichtlinien und -standards im Unternehmen 22
3.2 Industriell eingesetzte Software-Testverfahren 24
3.2.1 Allgemeine Aspekte des Software-Testens 24
3.2.2 Testverfahren in der Softwareentwicklung 24
3.2.3 Testverfahren im Bereich Automotive 33

4 Qualitätssicherungsverfahren akademischen Ursprungs **38**
4.1 Verifikationsmethoden 38
4.2 Modelchecking 39
4.2.1 Modellierungssprache 40
4.2.2 Spezifikationsprache 42
4.2.3 Verifikationsalgorithmus 43

5 Fallbeispiel: Türsteuergerät **45**
5.1 Einsatz des Türsteuergerätes im Automobil 45
5.1.1 Funktionen des Türsteuergerätes 45

5.1.2 Hardwareeigenschaften 47
5.2 Türschloss 48
5.2.1 Verriegeln 48
5.2.2 Entriegeln 48
5.3 Fensterheber 49
5.3.1 Fensteröffnen bzw. -schliessen 50
5.4 Beleuchtung 50

6 Eingesetzte Werkzeuge 52
6.1 ASCET-Werkzeugkette 52
6.1.1 ASCET-MD 53
6.1.2 ASCET-RP 56
6.1.3 ASCET-SE 56
6.2 Unterstützende Werkzeuge 57
6.2.1 IBM Rational Test RealTime 57
6.2.2 CTE XL 57
6.3 AutoFocus 58
6.3.1 System Structure Diagram (SSD) 59
6.3.2 State Transition Diagram (STD) 59
6.3.3 Extended Event Traces (EET) 60
6.3.4 Data Type Definitions (DTD) 61
6.3.5 Modelchecking-Anbindung 61

7 Modellierung in ASCET 64
7.1 Modell des Türsteuergerätes 64
7.1.1 Rahmenbedingungen 64
7.1.2 Aufbau und Funktionsweise 69
7.2 Testen auf Modellebene (Simulation) 77
7.3 Testen auf Codeebene 78
7.4 Rapid-Prototyping 80

8 Modellierung in AutoFocus 84
8.1 Modell des Türsteuergeräts 84
8.1.1 Rahmenbedingungen 84
8.1.2 Aufbau und Funktionsweise 86
8.2 Testverfahren 93
8.3 Modelchecking 95
8.3.1 Verifikation des Einklemmschutzes 97

9 Evaluierung 103

9.1 Bewertung der Werkzeuge 103
9.1.1 Modellierung 104
9.1.2 Zeitlicher Aufwand bei der Modellierung 105
9.1.3 Zusammenhang zwischen Modell und Code 108
9.2 Bewertung der Qualitätssicherungsmethoden 110

10 Fazit und Ausblick **117**

A Spezifikationssprachen **120**
A.1 CTL 120
A.1.1 Syntax 120
A.1.2 Semantik 120
A.2 LTL 121
A.2.1 Syntax 121
A.2.2 Semantik 121

B Hardware-Schnittstelle des Türsteuergerätes **123**
B.1 Stecker S1: Anschluss von Elementen der Vordertür . . . 123
B.2 Stecker S2: Anschluss von externen Elementen 124

C Modellierung in ASCET **126**
C.1 Benutzerdefinierte Datentypen 126
C.2 Nachrichten 127

Abbildungsverzeichnis **130**

Tabellenverzeichnis **132**

Literaturverzeichnis **133**

Kapitel 1

Einleitung

Sowie früher die Elektronik ihren Einzug in das alltägliche Leben gefunden hat, spielt heutzutage die Software in eingebetteten Systemen eine zunehmend wichtigere Rolle. Die Funktionalität eines Produktes wird in vielen Fällen massgeblich durch Software bestimmt. In vielen Industriezweigen stellt dies einen wettbewerbsentscheidenden Faktor dar.

Dies trifft ebenso auf den Automobilbausektor zu, der zudem durch hohe Sicherheitsanforderungen an die Komponenten eines Kraftfahrzeuges gekennzeichnet ist. Der ständig wachsende Anteil, die kurzen Entwicklungszyklen und die steigende Komplexität von Software im Automobil stellen hohe Herausforderungen an die Automobilhersteller dar. Diese Umstände werden durch den hohen Kostendruck und die zur Verfügung stehenden beschränkten Rechnerressourcen nicht gemildert.

Fehler in der Software sicherheitskritischer Anwendungen können zu Funktionsausfällen und somit zu hohen Kosten führen. Durch den intensiven Einsatz von Qualitätssicherungsmassnahmen versucht man dieses Risiko zu minimieren, um ein höheres Mass an Zuverlässigkeit, Sicherheit und Qualität zu gewährleisten.

Moderne Entwicklungsverfahren, wie die Modellierung mit grafischen Beschreibungsmitteln und die automatische Codegenerierung sollen ebenfalls zu dem gewünschten Ergebnis beitragen.

1.1 Aufgabenstellung

Ziel ist die Anwendung und der Vergleich von konventionellen Software-Qualitätssicherungsverfahren mit Modelchecking. Andere Verifikationsverfahren akademischen Ursprungs werden im Rahmen dieser Diplomarbeit nicht näher betrachtet. Der Vergleich erfolgt im Kontext der Automobilindustrie. Als Fallstudie dient eine textuell verfasste, öffentlich zugängliche Spezifikation eines Türsteuergerätes [PH02]. Die darin beschriebenen Komponenten für das Türschloss, die Fensterheberansteuerung und die Lichtansteuerung sollen mit Hilfe der beiden CASE-Werkezeuge ASCET und AutoFocus in Modelle umgesetzt werden. Darauf basierend sollen anschliessend Qualitätssicherungsmassnahmen wie Modelchecking und White-Box-Tests angewandt, bewertet und gegenübergestellt werden. Der Vergleich der Verfahren soll Vor- und Nachteile aufzeigen, die beim Einsatz von formalen Verfahren im Entwicklungsprozess aufkommen.

1.2 Struktur der Diplomarbeit

Das zweite Kapitel gibt einen einführenden Überblick über den Entwicklungsprozess von Software. Die Vorzüge modellbasierter Entwicklung und die Entwicklung von Software in der Automobilindustrie werden darin erläutert.

Im dritten Kapitel werden traditionelle Techniken für das Testen von Software vorgestellt. Unterschieden wird hier zwischen Qualitätssicherungsverfahren für IT-Systeme und Verfahren für Software in Kraftfahrzeug-Steuergeräten.

Das darauf folgende Kapitel gibt eine kompakte Einführung in die Grundlagen von Modelchecking.

Als Basis für die Anwendung der beschriebenen Qualitätssicherungsmethoden dient die Spezifikation des oben genannten Türsteuergerätes. Das fünfte Kapitel befasst sich daher mit dessen für die Diplomarbeit relevanten Teilen.

Im Anschluss daran beschreibt das nächste Kapitel die eingesetzten Werkzeuge, mit denen die Spezifikation des Türsteuergerätes jeweils um-

gesetzt wurde. Schwerpunkte bilden einerseits ASCET als industriell eingesetztes CASE-Tool und andererseits AutoFocus als Modellierungswerkzeug akademischen Ursprungs mit einer Modelchecking-Anbindung. Die Anbindung an weitere Werkzeuge zur Durchführung der Tests werden hier ebenfalls behandelt.

Die folgenden beiden Kapitel dokumentieren die Modellierung und Anwendung der Test- und Verifikationsverfahren auf das Steuergerät mit Hilfe der vorgestellten Werkzeuge.

Basierend auf den gesammelten Erkenntnissen erfolgt im neunten Kapitel die Bewertung der eingesetzten Methoden.

Kapitel 2

Aspekte der Software Entwicklung

Schon die in der Einleitung erwähnten Bemerkungen haben deutlich gemacht, dass der Software-Entwicklungsprozess von IT-Systemen nicht ohne weiteres direkt im Kontext der Automobilindustrie angewendet werden kann. Im Folgenden werden daher zunächst allgemeine Gesichtspunkte des klassischen Software Engineering aufgezeigt.

Moderne Entwicklungsmethoden, wie die der modellbasierten Entwicklung haben mittlerweile auch im Bereich Automotive Fuss gefasst. Darauf, sowie auf den Software-Entwicklungsprozess von Steuergeräten wird im Anschluss eingegangen, um den Zusammenhang zum Automotive Software Engineering herzustellen.

2.1 Grundlegende Begriffe

Software Engineering kann man als die „zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden Konzepten, Notationen und Werkzeugen für die arbeitsteilige, ingenieurmässige Entwicklung und Anwendung von umfangreichen Software-Systemen“ auffassen [Bal00]. Darunter fällt auch die Beschreibung von Vorgehensmodellen, mit deren Hilfe man den Herstellungsprozess eines Softwareproduktes strukturieren kann. Der Entwicklungsprozess lässt sich dabei in mehrere grundlegende Aktivitäten zerlegen:

Anforderungen an das Endprodukt werden zunächst in der Analysephase berücksichtigt. Anschliessend erfolgt in der Entwurfsphase und in der Implementierungsphase die Umsetzung. Die Testphase, die Inbetriebnahme und die Wartung runden die Vorgehensweise ab.

Abhängig vom gewählten Vorgehensmodell fällt die Einteilung in diese Phasen und die Beziehungen zwischen ihnen unterschiedlich aus.

Ein Beispiel für ein konkretes Vorgehensmodell stellt das V-Modell dar. Anhand dessen für die Software-Erstellung zuständigen Teil werden die Beziehungen zwischen den einzelnen Phasen veranschaulicht:

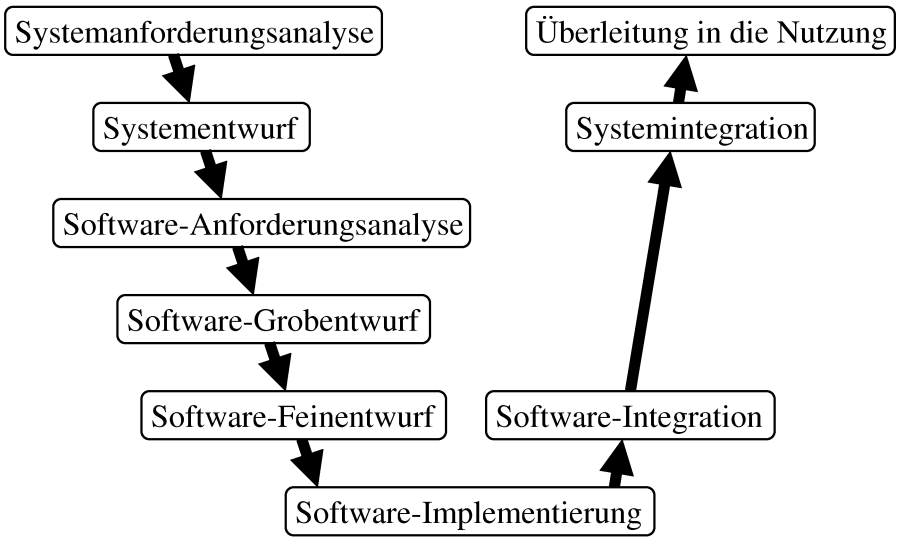


Abbildung 2.1: Phasen der Software-Entwicklung.

In Form eines textuell verfassten Lastenheftes werden die Anforderungen auf informelle Art und Weise aus der Sicht des Auftraggebers formuliert. Die Umsetzung in einen Entwurf und die Implementierung erfolgt auf dem linken Ast in Bild 2.1. Dabei werden grosse Systemen zur besseren Beherrschbarkeit hierarchisch in Subsysteme zerlegt. Das Testen der Komponenten und des zusammengesetzten Systems erfolgt auf dem rechten Ast.

Falls beim Testen im rechten Ast Fehler gefunden wurden, erfolgt die Korrektur rückwirkend an der entsprechenden Stelle im linken Ast. Die Phasen werden dann ab der Korrekturstelle nochmals durchlaufen. Es ergibt sich somit ein Zyklus, dessen Länge von der Fehlerstelle abhängt.

Je später, d.h. je weiter oben im rechten Ast, der Fehler gefunden wurde, desto aufwendiger wird dessen Korrektur. Oftmals findet die Implementierung eines Produkts zudem bei einem Zulieferer statt, so dass komplexe Zuliefererstrukturen die Fehlerbehebung zeitlich in die Länge ziehen können. Dies wirkt sich negativ auf die Fehlerbehebungskosten aus.

Gründe für Fehler kann man klassifizieren. Es ergeben sich unter anderem folgende Fehlerklassen:

1. Das Verhalten des Produktes wurde in der Spezifikation nicht eindeutig formuliert. Der Interpretationsspielraum hat zur falschen Umsetzung geführt.
2. Das Verhalten des Produktes wurde in der Spezifikation nicht vollständig beschrieben. Die Ergänzung des Verhaltens nach eigenen Vorstellungen war fehlerhaft.
3. Der Grund für den Fehler war programmiertechnischer Natur.

In diesem Zusammenhang soll der Unterschied zwischen den beiden Begriffen „Verifikation“ und „Validierung“ erläutert werden. Unter „Verifikation“ versteht man die Überprüfung, ob die vorgegebene (technische) Spezifikation korrekt in das Produkt umgesetzt worden ist. „Validierung“ hingegen bezeichnet die Überprüfung, ob das Produkt die gestellten Anforderungen überhaupt erfüllt bzw. ob es sich für seinen vorgesehenen Einsatzzweck eignet. Man kann dabei funktionale Anforderungen, die auf den Zweck des Systems ausgerichtet sind, von nichtfunktionalen Anforderungen, wie Leistung, Laufzeit und Speicherverbrauch, unterscheiden.

Verifikation: *Wurde das Produkt richtig implementiert?*

Validierung: *Wurde das richtige Produkt implementiert?*

Die Verifikation befasst sich mit Fehlern der Fehlerklasse 3. Die Validierung adressiert Fehler aus den Fehlerklassen 1 und 2.

Somit stellen Verifikation und Validierung, z.B. in Form von Tests, qualitätsverbessernde Massnahmen dar. Organisatorisch werden diese im Rahmen der Testplanung berücksichtigt. Darin erfolgt die Zuteilung der Ressourcen, wie Mitarbeiter und der zeitlichen Einteilung der Tätigkeiten. Desweiteren wird festgelegt, mit welcher Intensität und Priorität die einzelnen Systemteile zu testen sind. Auch die zu verwendenden Werkzeuge werden hier bestimmt. Kapitel 3 geht auf das Testen

an sich noch genauer ein.

2.2 Modellbasierte Entwicklung

Wie oben angedeutet ist ein Grund für das Entstehen von Fehlern nicht nur eine missverständliche bzw. unvollständige textuelle Spezifikation. Dies liegt in der inhärenten Mehrdeutigkeit der menschlichen Sprache begründet. Verbessern kann man diese Situation durch die Strukturierung der Informationen mit Hilfe von CARE-Werkzeugen[1], wie z.B. Telelogics DOORS [Tel]. Einen weiteren Ausweg bilden formale Beschreibungstechniken, die in Kapitel 4 genauer erläutert werden. Damit lassen sich Spezifikationen mit Hilfe von Modellen beschreiben [SPW04]. Vorteile ergeben sich daraus durch die Einschränkung des Interpretationsspielraumes. Die internen Zusammenhänge eines formalen Modells können iterativ verfeinert, d.h. genauer spezifiziert werden. Modelle lassen sich dadurch soweit konkretisieren, dass sie als ausführbares Programm umgesetzt werden können. Sie erlauben somit eine frühzeitige Validierung gegenüber den eingangs gestellten Anforderungen des Auftraggebers.

Eine iterative Verfeinerung des Modells bis hin zur Implementierungsebene führt zu einem ineinandergreifenden Übergang der Entwicklungsphasen. Unterstützung hierbei leisten CASE-Werkzeuge[2] (siehe Kapitel 6). Mit ihrer Hilfe lassen sich Funktionsmodelle zunächst ohne detaillierte Programmierkenntnisse erstellen. Experten aus der Anwendungsdomäne können somit das Verhalten spezifizieren, ohne sich Gedanken über konkrete implementationstechnische Einschränkungen (Datentypenkonvertierungen, etc.) machen zu müssen. Zudem können Simulationen und Rapid Prototyping (Kapitel 7.4) die Anforderungen absichern. Mehrdeutigkeiten und fehlende Beschreibungen im Lastenheft können somit ausgemacht werden. Dies lässt ein frühzeitiges Erkennen und Beseitigen von Fehlern zu, ohne den gesamten Fehlerbehebungszyklus (siehe Kapitel 2.1) nochmals durchlaufen zu müssen.

In der Implementierungsphase erfolgt die Übersetzung der Modelle in Code. Die manuelle Umsetzung der Modelle von Hand ist fehlerträchtig und die Absicherung des Codes gegenüber dem Modell zeitintensiv. Der

[1] CARE steht für **C**omputer **A**ided **R**equirements **E**ngineering
[2] CASE steht für **C**omputer **A**ided **S**oftware **E**ngineering

Einsatz von Codegeneratoren behebt diese Schwäche. Ein Codegenerator erzeugt aus einem Modell den entsprechenden Code, dessen Kompilat dann auf der Zielmaschine zur Ausführung gebracht werden soll. Im Gegensatz zur Entwicklung auf Basis einer textuellen Spezifikation entfallen damit, auf Grund der automatisierten Werkzeug-Unterstützung, fehlerträchtige manuelle Arbeitsschritte (vgl. Bild 2.2, [Gmb03c]). Daraus ergibt sich eine erhöhte Produktivität und eine verbesserte Softwarequalität.

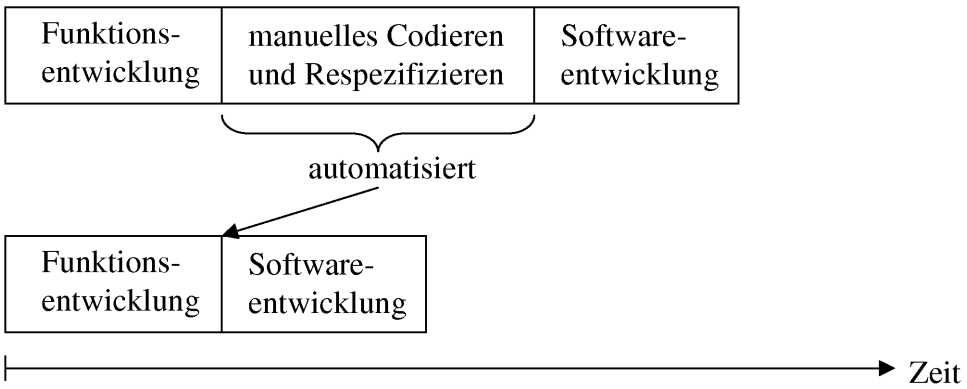


Abbildung 2.2: Verteilung des Arbeitsaufwandes.

Voraussetzung für ein fehlerfreies Ergebnis ist die korrekte Funktionsweise des Codegenerators und Compilers. Diversifizierende Testansätze (siehe Kapitel 3.2.3) und die Verwendung zertifizierter Codegeneratoren können hierbei die Fehlerwahrscheinlichkeit verringern.

Ein weiterer wichtiger Gesichtspunkt bei der Codeerzeugung im Umfeld von eingebetteten Systemen ist die Codeoptimierung. Auf Grund der kanonischen Umsetzung von Modellen in Code kann auf den Zeit- und Speicherverbrauch bei der automatischen Codegenerierung nicht so effizient eingegangen werden, wie bei der manuellen Programmierung von Hand. Auf lange Sicht kann dieser Nachteil jedoch vernachlässigt werden, da die Leistung und Ressourcen zukünftig eingesetzter Hardware tendenziell zunehmen.
Bei ausreichend vorhandenen Ressourcen wiegen die genannten Vorteile der Codegenerierung auf, was einen weiteren Grund für den Einsatz der modellbasierten Entwicklung darstellt.

Als konkretes Beispiel für eine Werkzeugkette, das einen modellbasierten Entwicklungsprozess unterstützt, sei hier die SCADE-Suite erwähnt

[Gä06]. Die Firma Esterel Technologies [Est] stellt damit eine modellbasierte Entwicklungsumgebung mit zertifiziertem Codegenerator zur Verfügung.

2.3 Software-Entwicklung im Bereich Automotive

Randbedingungen, wie die eingeschränkten Ressourcen, die hohen Sicherheits- und Zuverlässigkeitsanforderungen, erfordern ein anderes Vorgehen bei der Entwicklung von Software für das Automobil im Gegensatz zur traditionellen Softwareentwicklung von IT-Softwaresystemen. Im folgenden soll deswegen auf einige Konzepte und technische Begriffe, die in der Automobilindustrie Anwendung finden, eingegangen werden.

Die Regelung und Steuerung von physikalischen Vorgängen im Kraftfahrzeug erfolgt in zunehmendem Masse durch Steuergeräte. Ein Steuergerät bzw. eine ECU[3] ist ein elektronisches Modul in Form eines eingebetteten Systems, dessen Funktionalität durch die darauf laufende Software bestimmt ist. Über die Anbindung an Sensoren und Aktuatoren kann das Steuergerät als Teil eines Regelkreises, bestehend aus Sollwertgeber, Regler, Aktuatoren, Strecke und Sensoren betrachtet werden:

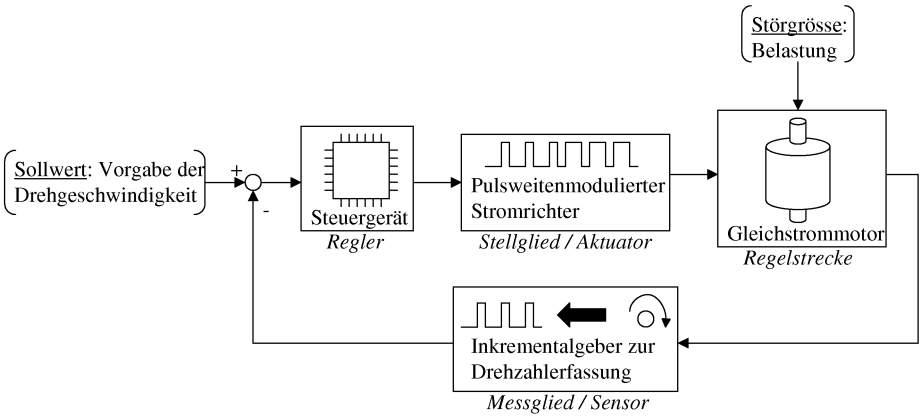


Abbildung 2.3: Das Steuergerät als Teil eines Regelkreises.

[3]ECU steht für **E**lectronic **C**ontrol **U**nit

Ein Beispiel für einen Sollwertgeber ist die Vorrichtung zur Übermittlung der Fahrpedalstellung. Beispiele für Sensoren sind Drehzahlmesser und Spannungsmesser. Aktoren können unter anderem Motoren oder Ventile sein. Die Strecke stellt den zu regelnden physikalischen Vorgang dar (z.B. Umdrehungsgeschwindigkeit). Das Steuergerät nimmt als Regler seinen Platz im Regelkreis ein. Durch die Verwendung von Steuergeräten lassen sich oft mehrere Steuer- und Regelungsfunktionen in einem Gerät vereinen, was zu Kosteneinsparungen führt.

Eine weitere Betrachtungsweise von Steuergeräten ergibt sich bei Berücksichtigung der Vernetzung der Steuergeräte untereinander. Die Kommunikation zwischen den Steuergeräten erfolgt dabei über ein Bussysteme, wie den CAN-Bus. Der CAN-Bus ist ein nachrichtenorientiertes Bussystem. Eine Nachricht beinhaltet ein Datenfeld, in dem mehrere Signalbits übertragen werden können. Funktionen, wie z.B. die Zentralverriegelung, lassen sich durch die Verwendung von Bussystemen räumlich auf die verschiedenen Steuergeräte verteilen. Der Aufwand für die Verkabelung fällt dadurch weit geringer aus. Hinzu kommt die einfachere Erweiterbarkeit und Variantenbildung durch den modularen Aufbau. Eine gesteigerte Zuverlässigkeit und Sicherheit kann durch eine ausfalltolerante Auslegung und Überwachungsrechner erreicht werden. Nachteilig ist jedoch das mit der Vernetzung einhergehende Problem der „Feature Interaction“. Eine Feature Interaction ist eine Situation, in der sich mehrere Funktionen ungewollt gegenseitig beeinflussen und dabei unerwartete negative Effekte hervorrufen.

Das gesamte vernetzte System von elektronischen Steuergeräten in einem Fahrzeug lässt sich in mehrere Subsysteme untergliedern. Man unterscheidet Antriebsstrang, Fahrwerk, Multi-Media und Karosserie, wobei sich letzteres Subsystem in Komfort und passive Sicherheit aufgliedern lässt. Beispiele für die Zuordnung von Funktionen zu Subsystemen:

Antriebsstrang	Motorsteuerung, Getriebesteuerung
Fahrwerk	Anti-Blockier-System
Karosserie (Komfort)	Zentralverriegelung, Spiegelverstellung, Fensterheber
Karosserie (passive Sicherheit)	Airbagsysteme
Multi-Media	Radio

Tabelle 2.1: Zuordnung von Funktionen zu Subsystemen.

Das Zusammenspiel der verschiedenen Komponenten und der verschiedenen Fachdisziplinen spiegelt sich in der Vorgehensweise bei der Entwicklung von elektronischen Systemen im Fahrzeug wieder. Hinzu kommt die firmenübergreifende Zusammenarbeit zwischen Automobilherstellern und Zulieferern. Die Entwicklung von elektronischen Systemen und Software gliedert sich in einen „Kernprozess“ und „Unterstützungsprozesse“. Folgendes Bild veranschaulicht den Kernprozess [SZ04]:

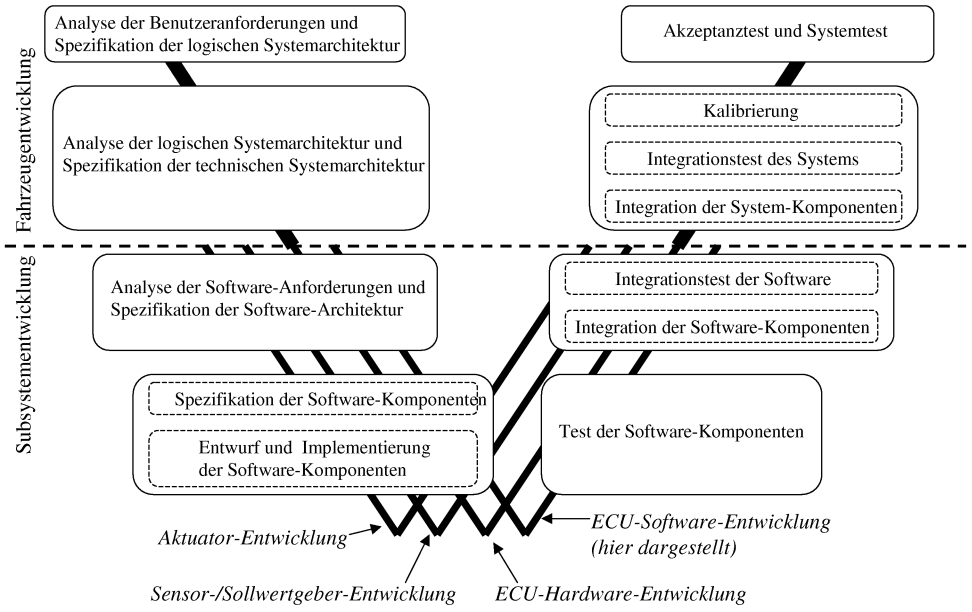


Abbildung 2.4: Kernprozess der Steuergeräte-Entwicklung.

Ausgehend von der Analyse der Benutzeranforderungen wird die „logische Systemarchitektur“ spezifiziert. Diese legt die einzelnen Funktionen,

deren Schnittstellen und die Kommunikation untereinander fest. Ein konkretes Beispiel für eine Funktion ist die Zentralverriegelung. Danach erfolgt die Abbildung der logischen Systemarchitektur (Funktionsnetzwerk) auf die „technische Systemarchitektur“ (Steuergerätenetzwerk). Dabei werden die einzelnen Funktionen den vorhandenen Steuergeräten zugeordnet. Im Falle der Zentralverriegelung sind die Türsteuergeräte davon betroffen (siehe Kapitel 5). Nachdem die technische Architektur für das System feststeht, erfolgt dessen Partitionierung in Steuergerätehardware, Steuergerätesoftware, Sollwertgeber, Sensoren, Aktuatoren und deren parallele Entwicklung. Die vorliegende Diplomarbeit befasst sich vorwiegend mit der Entwicklung der Software von Steuergeräten, weswegen nur der Teilast für die Software-Entwicklung im Bild detaillierter dargestellt ist. Nach der Implementierung wird das System wieder aus den einzelnen Teilsystemen zusammengesetzt. Komponenten-, Integrations-, Systemtests verifizieren das jeweilige Resultat bzgl. dessen Spezifikation. Eine Validierung gegenüber Benutzeranforderungen findet im Akzeptanztest statt. Die nachträgliche Kalibrierung nach der Integration der Systemkomponenten sorgt für eine Anpassung (regelungstechnischer) Parameter. Zum Beispiel können hier die Kennfelder eines Motorsteuergerätes für einen konkreten Motor individuell eingestellt werden.

Der Kernprozess wird von Unterstützungsprozessen begleitet, deren Aufgabengebiete in folgender Tabelle zusammengefasst sind [SZ04]:

Projektmanagement	Kosten-, Termin-, Organisationsplanung, Risikoanalyse
Konfigurationsmanagement	Varianten-, Versionsverwaltung
Anforderungsmanagement	Erfassen und Verfolgen von Anforderungen
Lieferantenmanagement	Verwalten der firmenübergreifenden Zusammenarbeit
Qualitätssicherung	Massnahmen zur Qualitätsprüfung und -sicherstellung

Tabelle 2.2: Unterstützungsprozesse.

Die enge Verzahnung der verschiedenen Prozesse untereinander ist an Hand der Qualitätssicherung ersichtlich. Neben Richtlinien kommen hier auch Massnahmen zur Anwendung, die sich im rechten Ast in Bild 2.1 in Form von Tests wiederfinden. Kapitel 3 geht nochmals näher darauf ein.

Einen weiteren wichtigen Schritt zur Beherrschung der steigenden Komplexität von elektronischen Geräten im Automobil stellt die Einführung von standardisierten Schnittstellen dar. Dazu haben sich namhafte internationale Automobilhersteller und Zulieferer zum AUTOSAR[4] Konsortium zusammengeschlossen [AUTb]. Ziel ist die Erarbeitung einer einheitlichen Software-Infrastruktur und von Schnittstellen in Form von Standards. Die Software kann dadurch modularer gestaltet werden. Besser ausgetestete Software-Komponenten können somit vermehrt wiederverwendet werden.

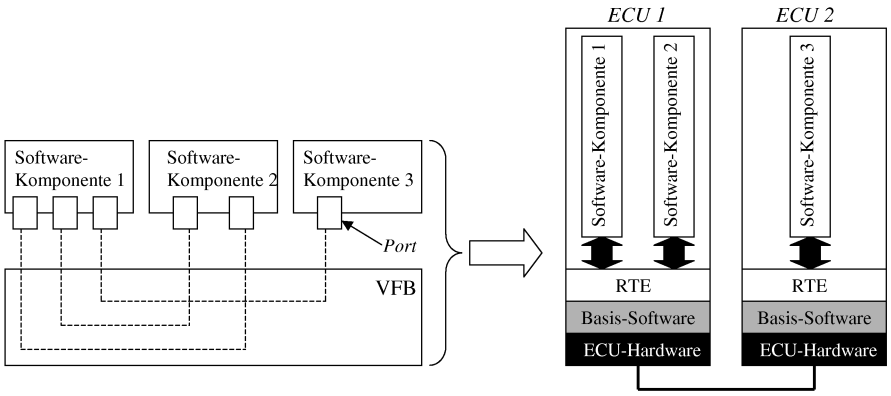


Abbildung 2.5: AUTOSAR Software-Architektur.

Hardwarespezifische Details sollen den Anwendungen verborgen bleiben, so dass diese unabhängig von der Hardware und portabel implementiert werden können. Auf individuelle Kundenwünsche auf Applikationsebene kann so leichter eingegangen werden. Zum Beispiel läuft die Kommunikation zwischen Software-Komponenten für diese transparent über Ports ab, so dass sie unabhängig vom Deployment auf dieselbe Schnittstelle zugreifen. Die Implementierung dieses „Virtuellen Funktionsbusses“ (VFB) erfolgt mit Hilfe des „AUTOSAR Runtime Environments“ (RTE), welches die Intra- und Inter-Steuergerätekommunikation verwaltet. Das RTE wiederrum setzt auf der Basis-Software auf, die den Zusammenhang zur Hardware eines Steuergerätes darstellt. Als Teil der Basis-Software kann ein OSEK-kompatibles Betriebssystem zum Einsatz kommen. OSEK-kompatible Betriebssysteme unterstützen den ISO 17356 Standard. Im folgenden wird dafür stellvertretend der Begriff OSEK verwendet, ohne sich auf ein bestimmtes OSEK-Derivat zu beschränken.

[4]AUTOSAR steht für **AUT**omotive **O**pen **S**ystem **AR**chitecture

Genauso wie bei AUTOSAR, muss OSEK zunächst konfiguriert werden, bevor es erstellt und eingesetzt werden kann. OSEK ist ein statisches Betriebssystem. Zur Laufzeit findet zum Beispiel keine dynamische Speicherallokation statt, denn die Anzahl der Anwendungen und eingesetzten Betriebsmittel wird fest vorkonfiguriert.

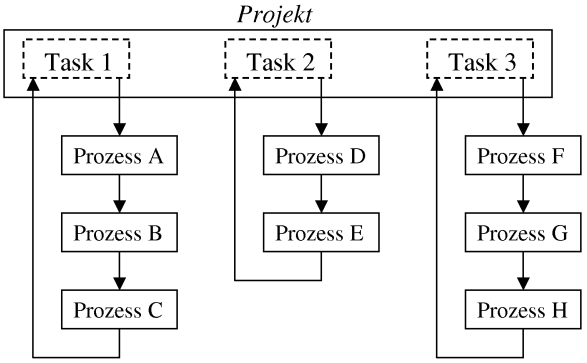


Abbildung 2.6: Zuordnung von Prozessen zu Tasks.

In OSEK werden Anwendungen auf Tasks und Prozesse aufgeteilt. Die Berechnungen finden in Prozessen statt, die wiederum zu Tasks gruppiert werden können. Innerhalb einer Tasks werden die Prozesse sequentiell hintereinander abgearbeitet. Auf einem Prozessor laufen mehrere Tasks quasi-parallel. Welche Task wann zur Ausführung kommt, hängt vom vorkonfigurierten Scheduling ab und beeinflusst das Echtzeitverhalten des Systems.

Kapitel 3

Konventionelle Qualitätssicherungsverfahren

Die Betonung in diesem Kapitel liegt auf Qualitätssicherungsmassnahmen, die in der Industrie weit verbreitet sind.

3.1 Qualitätssicherungsrichtlinien und -standards im Unternehmen

Ziel von Standards zur Qualitätssicherung ist die Erhöhung und Gewährleistung der Software-Qualität. Dem Stand der Technik angepasste Prozesse und Methoden definieren einheitliche Anforderungen an die Software-Entwicklung. Sicherheitsstandards und -richtlinien, wie in der Luftfahrtindustrie (z.B. DO-178B [RCT]) existieren ebenfalls in der Automobilindustrie. Ein Beispiel ist der IEC 61508 Standard [Com]. Er stellt Anforderungen an die Entwicklung von sicherheitskritischen elektrischen, elektronischen und programmierbaren elektronischen Systemen zur Erhöhung der Sicherheit. Es werden darin vier Sicherheitsstufen (SIL[1]) definiert. Ein SIL stellt ein Mass für die Wahrscheinlichkeit dar, mit der ein Sicherheitssystem die geforderten Sicherheitsfunktionen innerhalb eines Zeitraumes erfüllen kann. SIL 4 stellt die höchste Stufe mit geringer zulässiger Ausfallswahrscheinlichkeit dar und SIL1 die niedrigste

[1]SIL steht für **S**afety **I**ntegrity **L**evel

Stufe mit höherer zulässiger Ausfallswahrscheinlichkeit.
Unter anderem basieren unternehmensspezifische Standards auf dem generischen IEC 61508 Standard. Dabei werden oft eigene Abstufungen verwendet, wie z.B. der „Automotive-Sicherheits-Integritätslevel“ (ASIL) statt SIL. Abhängig vom ASIL eines betrachteten Testobjekts eignen sich die verschiedenen Qualitätssicherungsverfahren (siehe Abschnitt 3.2) unterschiedlich gut. In einem unternehmensspezifischen Standard kann somit, abhängig vom konkreten ASIL, eine Auswahl von Testmethoden für den Einsatz im eigenen Projekt vorgegeben werden [FTWT+05]. Daneben berücksichtigen Standards oft organisatorische Massnahmen und spezifizieren somit die Vorgehensweise für einen systematischen Testablauf und die Testplanung.

Im Allgemeinen geben Standards den Rahmen, aber nicht Details für die Vorgehensweise vor. Dies erlaubt eine individuelle Anpassung an das jeweils eigene Projekt. Unternehmensspezifische Leitlinien bieten sich hier als Entscheidungs- und Orientierungshilfen bei der technischen Umsetzung an. Im Gegensatz dazu geben Richtlinien einzuhaltende Regeln und Vorgaben an. Sie stellen im Kontext der Qualitätssicherung vorbeugende Massnahmen dar, die die Fehlerwahrscheinlichkeit im voraus verringern sollen. Unternehmensspezifische Richtlinien existieren unter anderem für die Entwicklung auf Modellebene mit Hilfe von industriell eingesetzten Modellierungswerkzeugen [Hua04, Mei05]. Sie definieren die Struktur eines Modells, um die Qualität der Modelle und der zu entwickelnden Software hinsichtlich Funktionalität, Zuverlässigkeit, Benutzbarkeit, Änderbarkeit und Wiederverwendbarkeit sicherzustellen. Damit sollen Flüchtigkeitsfehler und typische Fehler verhindert werden. Beispiele für Vorgaben sind:

- Übersichtliche Modellierung durch Einhalten von Grenzen und Abständen
- Verwenden von Namenskonventionen, indem aussagekräftige Bezeichner für Komponenten gewählt werden
- Initialisierungspflicht für Variablen

Desweiteren gibt es Codierungsrichtlinien zum Verhindern von Fehlern auf Sourcecodeebene. Die MISRA-C Richtlinie [MIS98] ist ein Beispiel dafür. Beispielregeln dienen zur Vermeidung von *goto*-Anweisungen oder von Zeigerarithmetik im C Code. Auch manche Codegeneratoren (vgl.

Abschnitt 2.2) erzeugen MISRA-C-konformen Code [Est].

3.2 Industriell eingesetzte Software-Testverfahren

3.2.1 Allgemeine Aspekte des Software-Testens

Gemäss [Mye01] verfolgt man beim Testen folgendes Ziel:

> *„Testen ist der Prozess, ein Testobjekt mit der Absicht zu überprüfen, Fehler zu finden".*

Testen stellt somit einen destruktiven Prozess dar, der psychologisch nicht im Einklang mit der Tätigkeit eines Entwicklers steht. Denn aus der Sicht eines Testers war der Test erfolgreich, falls ein Fehler aufgetreten ist. Einen Entwickler demotiviert diese Einstellungsweise. Aus seiner Sicht würde er damit beabsichtigt seine Arbeit degradieren. Damit die Effizienz des Testprozesses davon nicht beeinflusst wird, sollte die Aufgabe des Testens nicht von dergleichen Entwicklergruppe, sondern einer separaten Testergruppe durchgeführt werden.

3.2.2 Testverfahren in der Softwareentwicklung

Traditionelle Software-Tests basieren auf statischen und dynamischen Verfahren (siehe Bild 3.1). Statische Verfahren überprüfen ein Testobjekt an sich, ohne dessen Ausführung. Darunter fallen unter anderem formale (vgl. Kapitel 4) und informelle Verfahren, die auf den zur Verfügung gestellten Dokumenten (z.B. C-Code oder Modell) basieren. Bei letzteren Verfahren wird der Code in einer Diskussionsrunde besprochen, um Fehler aufzudecken, was sich nur für überschaubare Systeme bzw. Teile davon eignet. Die dynamischen Testverfahren bilden eine weitere Testgruppe. Hierbei wird das Testobjekt zur Ausführung gebracht. Das Testobjekt stellt im vorliegenden Fall eine Funktion oder ein System dar. Beim Testen werden Testfälle auf das Testobjekt angewandt und jeweils überprüft, ob bei der Ausführung ein Fehler aufgetreten ist. Die Testfälle werden von der Spezifikation des Testobjekts abgeleitet.

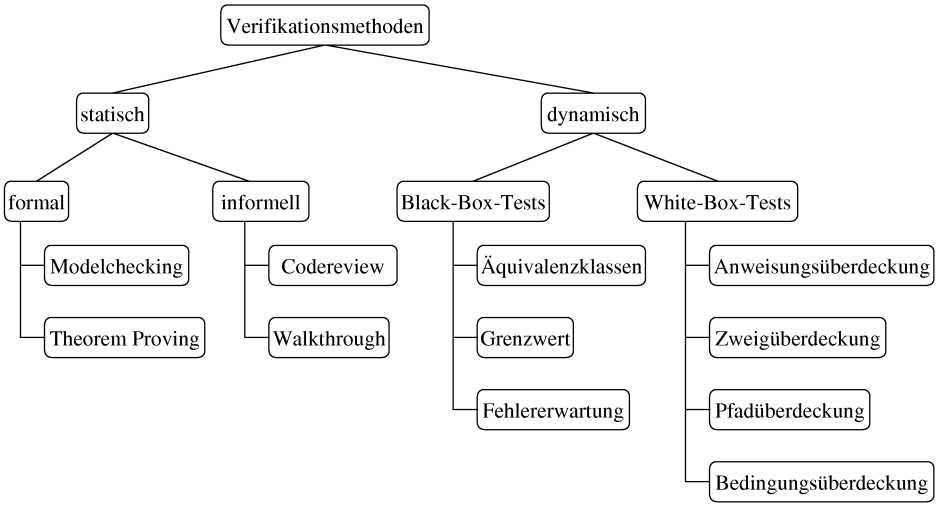


Abbildung 3.1: Überblick über gängige Testmethoden.

Ist das Testobjekt eine berechnende Funktion, so besteht ein Testfall aus konkreten Argumenten für die Funktion als Eingabe und der erwarteten Ausgabe. Ein Fehler tritt zum Beispiel auf, wenn die erwartete Ausgabe der Funktion nicht mit der aktuellen Ausgabe übereinstimmt.

Im Gegensatz zu einer Funktion erfolgt bei einem kompletten Softwaresystem im Allgemeinen nicht nur eine einmalige Auswertung von Eingabedaten (vgl. Abschnitt 8.2). Hier besteht ein Testfall aus einer Sequenz von Eingabewerten und den zugehörigen erwarteten Ausgabewerten.

Beim Testvorgang werden mehrere Testläufe nacheinander ausgeführt. Dabei wird je Testlauf ein Testfall auf das Testobjekt angewandt und die dabei entstandene Ausgabe mit der erwarteten Ausgabe verglichen. Wenn beide Ausgabewerte voneinander abweichen, deutet die Inkonsistenz auf einen Fehler im Testfall oder im Testobjekt hin.

Die 100%-ige Abdeckung des Testobjekts durch alle möglichen und erdenkbaren Testfälle ist im Allgemeinen unmöglich. Daher erfolgt das Testen nur solange, bis eine vorher festgelegte Bedingung eintritt. Das Erfüllen eines informell spezifizierten Ziel (z.B. Ausfall von mindestens zwei Komponenten) oder eine Abschätzung des Anteils noch vorhandener Fehler können dabei als „Testendekriterien“ dienen [RJdlCV04]. Ein weiteres oft eingesetztes Testendekriterium ist das Erreichen einer gewissen Überdeckungsgrösse, als Mass für den Grad der Vollständigkeit

eines Tests (siehe Kapitel 3.2.2). Abzusehen ist von zeitlichen Limits als Abbruchkriterien, da diese oft nicht mit der Fehlererkennungsrate korrelieren. Für den Fall, dass der Zeitplan für das Testen dennoch nicht eingehalten werden kann, sieht die Testplanung eine Testpriorisierung vor. Wichtigere Tests werden dann bei zeitlichen Engpässen vorgezogen.

Dynamische Testverfahren können weiterhin in Black-Box-Tests und White-Box-Tests untergliedert werden.

Black-Box-Tests

Bei der Betrachtung eines Objektes als „Black-Box“ ist dessen interne Struktur von aussen nicht sichtbar oder im betrachteten Kontext unbedeutend. Bei Black-Box-Tests betrachtet man folglich nur funktionelle Gesichtspunkte des Testobjekts. Zur Erstellung der Testfälle wird keine Information über den inneren Aufbau des Testobjekts herangezogen. Man greift hierbei auf die Spezifikation zurück.

Äquivalenzklassentests:

Ziel dieser Black-Box-Test-Variante ist es, eine hohe Fehlerentdeckungswahrscheinlichkeit mit einer minimalen Anzahl von Testfällen zu erreichen. Dabei werden die Wertebereiche der Eingabeparameter jeweils in Äquivalenzklassen unterteilt. Alle Elemente einer Äquivalenzklasse führen zum gleichen Verhalten des Testobjektes, d.h. man kann annehmen, dass Fehler, die mit einem Repräsentanten der Klasse[2] gefunden werden, ebenso mit allen anderen Elementen der Klasse gefunden werden können. Äquivalenzklassen sollten zur Repräsentation von gültigen, sowie von ungültigen Werten je Eingabeparameter erstellt werden. Das Erstellen von Testdaten erfolgt durch die Auswahl eines Repräsentanten je Eingabeparameter. Durch die Verwendung von Repräsentanten von Äquivalenzklassen je Eingabeparameter, statt allen Elementen des gesamten Eingabewertebereichs, kann man die Anzahl der Testfälle erheblich einschränken. Zeit- und Kosteneinsparungen im Testablauf sind die Folge.

Grenzwertanalyse:

Bei der Grenzwertanalyse werden gezielt die Randbereiche der Werte-

[2] beliebiges Element aus der Äquivalenzklasse

bereiche der Eingabeparameter untersucht, um dort Fehler aufzudecken. Man erhofft sich dadurch wirkungsvollere Testfälle. Die Grenzwertanalyse findet oft Anwendung in Kombination mit dem Äquivalenzklassentest. Als Eingabewert dient dann ein Repräsentant aus dem Randbereich der zugehörigen Äquivalenzklasse.

Intuitive Testfallermittlung (Fehlererwartung/error guessing):

Ziel dieser Art von Testfallerstellung ist die qualitative Verbesserung und Ergänzung bisheriger Testfälle, basierend auf Erfahrungswerten des Testers. Hier gibt es keine systematisches Vorgehensweise. Die Intuition des Testers im Fehlerfinden basiert auf einer langjährigen Erfahrung im Testen und im Umgang mit der Anwendungsdomäne.

Neben den hier erwähnten gibt es noch eine Fülle weiterer Black-Box-Test-Verfahren, die jedoch im Rahmen der Diplomarbeit nicht weiter betrachtet werden [Mye01].

White-Box-Tests

Bei der Betrachtung eines Objektes als „White-Box“ ist dessen interner Aufbau nach aussen hin sichtbar und analysierbar. Deswegen kann man im Gegensatz zu den vorgestellten Black-Box-Testverfahren bei der Testfallermittlung für White-Box-Tests auf die innere Struktur des Systems zurückgreifen. Der Ablauf von Berechnungen kann unter anderem mit Hilfe eines Kontrollflussgraphen visualisiert werden (siehe Abbildung 3.2).

Die Berechnungsschritte eines Programms werden dabei den Knoten eines gerichteten Graphen zugeordnet. Übergänge zwischen Berechnungsschritten, Sprünge und Verzweigungen im Programm werden durch Transitionen zwischen den Graphenknoten dargestellt. Ausgehend vom Startknoten werden dann entsprechend dem Berechnungsablauf des Programms die verschiedenen Knoten durchlaufen. Jeder Testfall entspricht somit einem Pfad durch den Kontrollflussgraphen. Dies ermöglicht Aussagen über den Grad der Abdeckung des Testobjektes beim Testen. White-Box-Tests können somit verwendet werden, um Testfälle, die durch Black-Box-Methoden gewonnen wurden, zu ergänzen.

Folgende Varianten von White-Box-Testverfahren unterscheiden sich in der Definition ihrer „Überdeckungskriterien“, als Mass für die Abdeckung

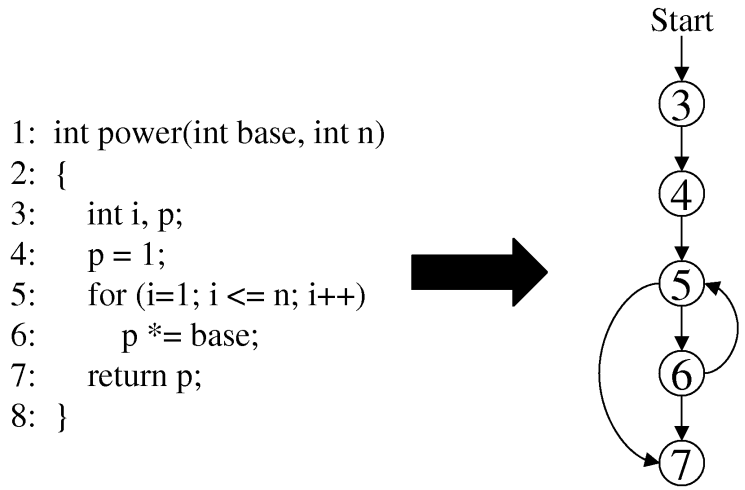


Abbildung 3.2: Beispiel für einen Kontrollflussgraphen.

durch Testfälle.

Anweisungsüberdeckungstests:

Bei diesem Testverfahren wird als Überdeckungskriterium folgendes Mass definiert:

C_0-Überdeckung $= \frac{Anzahl\ durchlaufener\ Anweisungen}{Gesamtzahl\ der\ Anweisungen} * 100\%$

Für die 100%-ige Abdeckung eine Programms gemäss dem eben definierten Mass müssen alle Anweisungen des Programms mindestens einmal ausgeführt werden.

Zweigüberdeckungstests:

An Verzweigungsstellen in einem Programm teilt sich der Kontrollfluss in mehrere Zweige auf. Hier versucht man alle Alternativen an den Verzweigungsstellen durch Testfälle abzudecken:

C_1-Überdeckung $= \frac{Anzahl\ durchlaufener\ Zweige}{Gesamtzahl\ der\ Zweige} * 100\%$

Bei einer 100%-igen Testabdeckung wird jeder Zweig an jeder Verzweigungsstelle mindestens einmal durchlaufen.

Pfadüberdeckungstests:

Bei der Ausführung eines Testfalles wird ein gesamter Pfad durch den Kontrollflussgraphen verfolgt. Das Überdeckungskriterium wird hier

demzufolge definiert als:

C_∞-Überdeckung $= \frac{Anzahl\ durchlaufener\ Pfade}{Gesamtzahl\ der\ Pfade} * 100\%$

Für eine 100%-ige Testabdeckung müsste jeder Pfad mindestens einmal durchlaufen werden. In der Praxis ist dies fast nie erreichbar, da *while*-Schleifen unter Umständen unendlich oft durchlaufen werden können. Daher dient die C_∞-Überdeckung eher als theoretisches Vergleichsmass.

Eine 100%-ige Abdeckung von Pfaden erreicht man unter Umständen mit Hilfe einer Abschwächung: Es werden dann nur Pfade betrachtet, deren *while*-Schleifendurchläufe an eine festgelegte Anzahl gebunden sind.

Bedingungsüberdeckungstests:

Die obigen Überdeckungskriterien waren vorwiegend kontrollflussorientiert. Daneben gibt es auch noch bedingungsbasierte Überdeckungskriterien. Diese betreffen die Bedingungen in einem Programm (z.B. an *if*-Entscheidungsstellen), die den Programmablauf beeinflussen. Eine zusammengesetzte Bedingung kann in atomare Ausdrücke zerlegt werden. Zum Beispiel besteht die C-Bedingung „aBool && ($bInt < cInt$ || !dBool)“ aus den drei atomaren Ausdrücken „aBool“, „$bInt < cInt$“ und „!dBool“. Für eine 100%-ige Abdeckung des Testobjekts existieren damit verschiedene Überdeckungskriterien, die beim Testen, je Bedingung im Programm, angewandt werden können:

1. *Einfache Bedingungsüberdeckung (branch condition testing)*: Jeder atomare Ausdruck muss mindestens einmal zu wahr bzw. falsch ausgewertet worden sein. Das Kriterium wird für oben genannen Beispielausdruck erfüllt, wenn während des Testablaufs folgende Auswertungen stattfinden[3]:

aBool	$bInt < cInt$	!dBool
0	0	0
1	1	1

Tabelle 3.1: Einfache Bedingungsüberdeckung

[3]jede Zeile entspricht einer Auswertung der gesamten Bedingung

2. *Mehrfachbedingungsüberdeckung (branch condition combination testing)*: Jede Kombination von atomaren Ausdrucksauswertungen muss mindestens einmal vorkommen. Für das Beispiel müssen beim Testen also folgende Auswertungen auftreten:

aBool	$bInt < cInt$	!dBool
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Tabelle 3.2: Mehrfachbedingungsüberdeckung

3. *Minimale Mehrfachbedingungsüberdeckung (modified branch condition decision testing)*: All diejenigen Kombination von Ausdrucksauswertungen mit folgender Eigenschaft müssen mindestens einmal vorkommen: Unterscheiden sich zwei Kombinationen im Wahrheitswert eines einzigen atomaren Ausdrucks und zugleich des Gesamtausdrucks, so müssen diese Kombinationen während des Testablaufs auftreten.

aBool	$bInt < cInt$	!dBool	aBool && ($bInt < cInt$ || !dBool)
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabelle 3.3: Minimale Mehrfachbedingungsüberdeckung

Durch einen Vergleich kann man feststellen, dass die erste Zeile von Tabelle 3.2 in Tabelle 3.3 fehlt. Diese Zeile ist auch nicht mehr erforderlich, da sie die hier erforderliche Eigenschaft nicht erfüllt: Denn bei alleiniger Änderung der Auswertung eines einzigen atomaren Ausdrucks (entweder „aBool“, „$bInt < cInt$“ oder „!dBool“) von 0 auf 1 ändert sich der Gesamtausdruck „aBool && ($bInt < cInt$ || !dBool)“ nicht.

4. *Modifizierte Bedingungs-/Entscheidungsüberdeckung (modified condition/decisiton coverage bzw. MC/DC)*: Entspricht der minimalen Mehrfachbedingungsüberdeckung mit der Einschränkung, dass für jede atomare Aussage (auf die die Eigenschaft der minimalen Mehrfachbedingungsüberdeckung zutrifft) nur ein Paar von Kombinationen aufzutreten braucht.

Aus Tabelle 3.3 können für den Ausdruck „aBool“ die Paare mit den Zeilennummern 1/5, 3/7 oder 2/6 in die Tabelle 3.4 übernommen werden. Denn auf jedes dieser Zeilenpaare trifft die Eigenschaft aus Punkt 3 zu. Hier fiel die Entscheidung zugunsten letzteren Paares. Analog wird für die anderen atomaren Ausdrücke „$bInt < cInt$“ und „!dBool“ auch ein Zeilenpaar aus Tabelle 3.3 ausgewählt und zu Tabelle 3.4 hinzugefügt.

aBool	$bInt < cInt$	!dBool	aBool && ($bInt < cInt$ || !dBool)
0	1	0	0
1	0	0	0
1	0	1	1
1	1	0	1

Tabelle 3.4: MC/DC

Weitere Testschritte

Ein System lässt sich oft hierarchisch zerlegen. Auf der untersten Hierarchieebene befinden sich die Module, auf die Tests mit den oben erwähnten Kriterien angewendet werden können. Daher werden diese Tests auch als *Modultests* bezeichnet. Im Sinne des IEC 61508 Standards kann eine C-Funktion eines C-Programmes als Modul aufgefasst werden. Oftmals müssen dabei mehrere Module miteinander interagieren, z.B. durch den

Aufruf einer anderen C-Funktion. Da der Testprozess zeitlich so früh wie möglich beginnen soll, liegen nicht immer alle Komponenten für einen reibungslosen Testablauf vor. Dann können eigens dafür entwickelte Testtreiber und -stubs die fehlenden Komponenten ersetzen. Testtreiber ersetzen die Schnittstelle zum darüberliegenden Teilsystem, in dem sich die betrachteten Komponenten befinden. Dies wäre z.B. eine main-Funktion, die die zu testende C-Funktion mit Eingabedaten speist. Mit Teststubs werden die fehlenden Komponenten (hier: die aufgerufenen Funktionen) emuliert. Durch diese Isolierung von Modulen, lassen sich diesen explizite Eingabewerte übergeben, die sie im integrierten System sonst nicht erhalten hätten. Damit lässt sich auch deren Verhalten bei undefinierten Eingabewerten testen.

Wenn sich während der Entwicklung Änderungen am Testobjekt ergeben, muss dieses nochmals überprüft werden. Dies erfolgt in den sogenannten *Regressionstests*. Daher ist es notwendig, bisher erstellte Testfälle aufzuzeichnen, um bei der erneuten Testfallermittlung Zeit zu sparen. Systeme, deren Anfangszustände einfach wieder hergestellt werden können, eignen sich für Regressionstests, da für jeden Testlauf die Anfangsbedingungen erneut eingerichtet werden können. Bei Steuergeräten ist diese Wiederholbarkeit gegeben.

Nachdem alle Module getestet wurden, können diese zu grösseren Teilsystemen zusammengesetzt werden. Der anschliessende *Integrationstest* befasst sich mit dem Finden von Fehlern beim Zusammenspiel der einzelnen Komponenten und bei den Schnittstellen zwischen den Komponenten. Der Aufwand, der für Integrationstests betrieben wird, äussert sich z.B. in der hohen Anzahl von Testfällen. Beim funktionalen Test von Fahrwerkssystemen von Mittelklasse-PKWs liegt deren Anzahl schon bei 200000 [WD05]. Automatisierte Testabläufe sind daher eine Voraussetzung, um diese Menge von Daten innerhalb einer beschränkten Zeit zu bewältigen. Abschliessend erfolgt auf der obersten Hierarchieebene der *Systemtest*. Hier erfolgt die Validierung gegenüber den Systemanforderungen. Gleiches gilt für den *Abnahmetest*, der jedoch vom Auftraggeber durchgeführt wird.

Die eben aufgeführten Schritte finden sich in Bild 2.1 auf Seite 12 auf dem rechten Ast wieder. Trotz der parallelen Vorgehensweise bei der Systemintegration im Entwicklungsprozess und im Testprozess sollten diese beiden Prozesse unterschieden werden. Denn konträr zur oben be-

schriebenen „Bottom-Up“-Strategie kann ein System zunächst in seiner Gesamtheit getestet werden, bevor rekursiv die Teilsysteme getestet werden („Top-Down“). Nichtexistente Teilsysteme werden hierbei mit Hilfe von Stubs emuliert.

Beim „Ad-Hoc“-Ansatz werden die Komponenten in einer zufälligen Reihenfolge getestet und zusammengefügt.

Eine nicht inkrementelle Vorgehensweise stellt der „Big-Bang“-Ansatz dar. Hier werden alle Komponenten erst einzeln getestet. Nachdem das gesamte System integriert ist, wird abschliessend nochmal getestet, ohne dazwischenliegende Integrationsschritte.

3.2.3 Testverfahren im Bereich Automotive

Die oben besprochenen Verfahren sind im Laufe der Jahre im Kontext von IT-Softwaresystemen entstanden. Mit dem zunehmenden Anteil von Software im Automobil, hat das Thema Software-Testen auch hier an Relevanz gewonnen. Allerdings kommen dabei dedizierte Testverfahren zur Anwendung, die die Software in den Kontext eines Steuergerätes einbeziehen. Der Zusammenhang zur Anwendungsdomäne und den anderen technischen Fachdisziplinen wird auch hier deutlich sichtbar.

Zum Beispiel fliessen im Kontext der Motorsteuerung Erfahrungen aus dem Maschinenbau und der Elektrotechnik ein. In einem grafischen Tool (siehe Kapitel 6) erstellt der Spezialist der Anwendungsdomäne ein regelungstechnisches Funktionsmodell, das die Zündung ansteuert, das Gasgemischt korrekt einstellt, etc. Stabilitätsbetrachtungen im Zusammenhang mit der Regelstrecke stehen hier im Vordergrund. Dieses Modell wird dann an die Software-Entwickler weitergereicht, welche wiederum die technischen Anpassungen an die Hardware, Umgebung und die Umsetzung in Code (manuell oder per Codegenerator) vornehmen. Beispielsweise erfolgt hier die Anpassung an unterstützte Datentypen etc. Durch die entsprechende Unterstützung durch CASE-Tools lässt sich dieser Prozess beschleunigen und die Umsetzung in Code automatisieren (vgl. Kapitel 2.2).

Zum Testen eines solchen Systems sind daher angepasste Verfahren notwendig, die auch den restlichen Kontext des Regelkreises miteinbeziehen, um ein korrektes Zusammenspiel zu gewährleisten. Das Testobjekt wird

in die entsprechende Umgebung eingebettet und dort getestet. Falls die Umgebung noch nicht existiert bzw. das Testobjekt noch nicht eingesetzt werden kann, können „X in the loop“-Systeme die fehlende Umgebung ersetzen. Abhängig vom aktuellen Stand der Entwicklung eines Systems hat „X in the loop“ eine andere Ausprägung [Har01]:

1. Model in the loop (MIL)

Bei Vorliegen des Funktionsmodells in einem CASE-Tool kann dieses simuliert werden. Die Umgebung wird ebenfalls modelliert und an das Funktionsmodell angebunden, um die Testdaten zur Verfügung zu stellen und die Rückkoppelung zu simulieren. Oben besprochene Testverfahren sind hier anwendbar, insofern die grafische Notation eine Überdeckungsanalyse zulässt. Als Vorteile der Simulation sind die schnell reproduzierbaren Ergebnisse zu nennen. Die Entwicklung und das Testen im Labor sparen damit Zeit und Kosten ein. Als Nachteil ergibt sich, dass manche Fehler unentdeckt bleiben, da sie nur unter realen Bedingungen eintreten. Denn auf Grund der Komplexität der phyikalischen Zusammenhänge kann die simulierte Regelstrecke nur eine Vereinfachung darstellen: Die Regelstrecke wird mittels Differentialgleichungen modelliert und im Arbeitspunkt linearisiert [MSF05].
Zudem ist eine Validierung gemäss Kundenwünschen nur in der realen Umwelt möglich, da nur dort ein endgültiger Funktionsnachweis erfolgen kann (beispielsweise bei der Erzeugung von authentischem Motorengeräusch oder der Bremspedalrückwirkung).

2. Software in the loop (SIL)

Nach der Umsetzung des Funktionsmodelles in Code, kann dieser auf der Entwicklungsplattform zur Ausführung gebracht werden. Oben besprochene Testverfahren sind hier ebenfalls anwendbar. Auch hier kann die Umgebung durch ein weiteres Software-System simuliert werden. Da das Gesamtsystem auf einer beliebigen leistungsstarken Hardware simuliert werden kann, spielen Echtzeiteigenschaften noch keine Rolle. Die Geschwindigkeit, mit der das Funktions- und Umgebungsmodell ablaufen, kann jeweils beliebig eingestellt werden. Analog zu MIL ist SIL frei von zeit- und kostenintensiven Versuchen.

3. Processor in the loop (PIL)

Hier wird der Code auf einem Testboard auf einem Prozessor ausgeführt, der der Zielplattform entspricht. Die Umgebung wird hier auf einem se-

paraten Rechner im Labor simuliert, der mit dem Testboard kommuniziert. Um obige Testverfahren der Code-Überdeckungsanalyse anzuwenden, muss der Code „instrumentiert“ werden. Dabei werden Zähler und weitere Hilfskonstrukte in den Code eingefügt. Bei der Ausführung des Kompilats mit bestimmten Testdaten werden die instrumentierten Stellen teilweise durchlaufen. Die Zählerstände geben am Ende Aufschluss über die Anzahl der Durchläufe. Durch das Instrumentieren verändert sich jedoch das Laufzeit- und Ressourcenverhalten des Testobjekts, was zu anderen Ergebnissen als bei MIL oder SIL führen kann.

4. Hardware in the loop (HIL)

Der Code wird für die Zielplattform generiert und in das Steuergerät geladen. Die Kopplung an den Rechner mit der Umgebungssimulation erfolgt über spezielle Hardware-Schnittstellen. Diese nehmen teilweise selbst die Rückkopplung vor, wenn eine reaktionsschnelle Signalverarbeitung gefordert ist. Auch eine Lastsimulation liegt daher in ihrem Aufgabengebiet.

Bei den eben beschriebenen „X in the loop“-Systemen wird eine nicht existierende Umgebung simuliert. Falls eine reale Umgebung zur Verfügung steht, lässt sich das Testobjekt darin einbetten. Einen populären Ansatz auf der Modellebene stellt das „Rapid-Prototyping“ dar. Ziel ist das frühzeitige Validieren von Funktionen am realen System. Das Modell wird mit Hilfe spezieller Rapid-Prototyping-Hardware, die an die Umgebung angebunden ist, ausgeführt. Voraussetzung ist das Vorliegen eines ausführbaren Modells (vgl. Abschnitt 2.2). In Abschnitt 7.4 wird näher darauf eingegangen.

Im Gegensatz zum Rapid-Prototyping liegt beim „Onboard“-Testsystem ein Prototyp-Fahrzeug mit realer Steuergeräte-Hardware vor, so dass die Steuergeräte-Software darauf getestet werden kann. Die Interaktion erfolgt über den Fahrer.

In folgender Tabelle sind obige Testsysteme nochmals zusammengefasst.

	Umwelt	
	simuliert	real
Modellebene	Model-in-the-loop	Rapid-Prototyping
Softwareebene	Software-in-the-loop	
Testboard	Processor-in-the-loop	
Steuergerät	Hardware-in-the-loop	Onboard Test

Tabelle 3.5: Klassifikation der Testsysteme.

Folgende Grafik veranschaulicht eine beispielhafte Einordnung der Testverfahren in den zeitlichen Ablauf der Entwicklung von Steuergeräte-Software im Zusammenspiel mit der Hardware-Entwicklung:

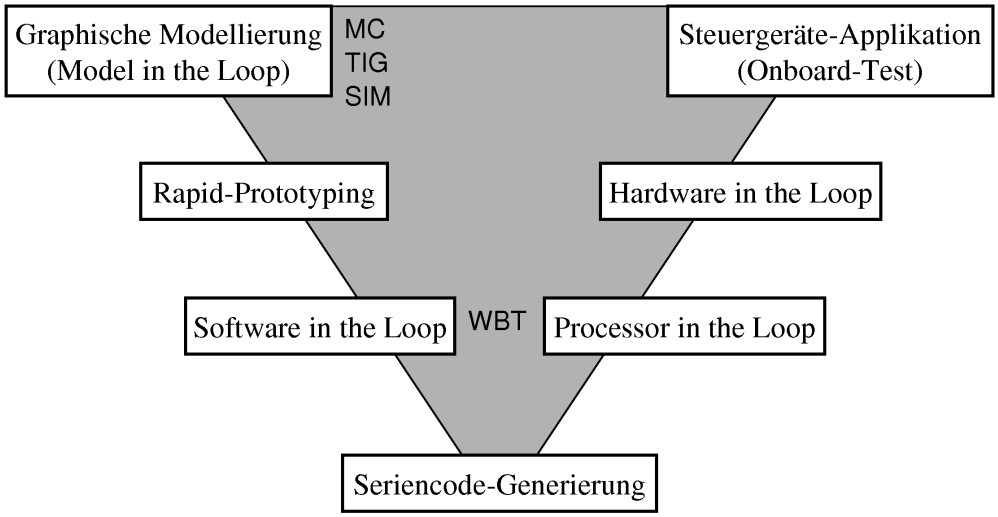


Abbildung 3.3: Einordnung der Testsysteme.

MC = Modelchecking (Abschnitt 8.3)
TIG = Modellbasierte Testfallgenerierung (Abschnitt 8.2)
SIM = Simulation (Abschnitt 7.2)
WBT = White-Box-Tests (Abschnitt 7.3)

Codegeneratoren nehmen, wie schon in Abschnitt 2.2 angesprochenen, eine zunehmend wichtigere Rolle bei der Seriencode-Generierung ein. Allerdings werden spezifische Anpassungen im Sourcecode teilweise immer

noch per Hand durchgeführt. Der Abgleich des Modells mit dem Sourcecode erfolgt über einen diversifizierenden Testansatz. Hierbei liegen unterschiedliche Versionen[4] des Testobjekts vor, die dieselbe Spezifikation erfüllen sollen. Alle Testfälle führen dann im fehlerfreien Fall bei allen Versionen zu identischen Resultaten. Auf diese Weise können Codegeneratoren bzw. die manuelle Codeerstellung überprüft werden:

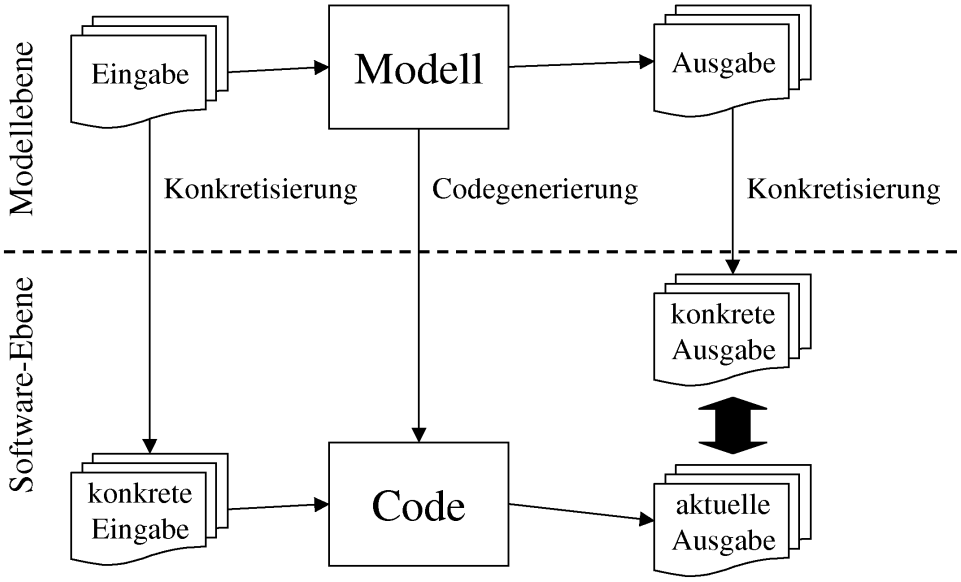


Abbildung 3.4: Diversifizierender Testansatz.

Das überprüfte und getestete Modell dient als Referenz. Dessen Testeingabe- und Testausgabedaten werden als Referenzplots aufgezeichnet. Die Testeingabedaten auf Modellebene werden in entsprechende Eingabedaten für die Software-Ebene umgesetzt (konkretisiert). Analog wird mit den Ausgabedaten des Modells verfahren. Aus dem Modell kann der Code generiert werden, der mit den eben konkretisierten Eingabedaten ausgeführt wird. Bei einer Abweichung der aktuellen Ausgabe von der konkretisierten Ausgabe kann man auf Fehler schliessen. Man nennt dieses Verfahren auch Back-to-Back-Test.

[4]hier: Modell bzw. Sourcecode

Kapitel 4

Qualitätssicherungsverfahren akademischen Ursprungs

Konventionelle Qualitätssicherungsverfahren zielen darauf ab, Fehler zu finden. Die Fehlerfreiheit von Software kann im Allgemeinen jedoch nicht mit traditionellen Testverfahren gezeigt werden. Dazu müsste die Korrektheit der Ausgaben zu allen möglichen Eingaben gezeigt werden. Praktisch ist dies nicht ohne erheblichen Aufwand durchführbar, da die 100%-ige Pfadüberdeckung des Testobjekts durch Testfälle (vgl. Abschnitt 3.2.2) oft nicht möglich ist. Dieses Defizit kann durch Verfahren akademischen Ursprungs behoben werden.

4.1 Verifikationsmethoden

Die, aus dem akademischen Bereich herrührenden Verfahren basieren auf der Anwendung formaler Methoden. Formale Methoden umfassen Vorgehensweisen zur Beschreibung und Analyse von Systemen, die auf mathematisch logischen Darstellungsmitteln und Verfahren beruhen. Beispielsweise lässt sich der Syntax von Beschreibungsnotationen eine widerspruchsfreie Semantik zuweisen. Die, auf dieser Grundlage erstellten „formalen Spezifikationen“ sind somit präzise definiert und haben eine mathematisch festgelegte Bedeutung. Damit können dann bestimmte Eigenschaften mathematisch gezeigt oder widerlegt werden. Ein Beispiel für eine formale Spezifikationstechnik ist FOCUS (siehe Abschnitt 6.3).

Die Verifikation im Sinne der Informatik ist der Nachweis, dass ein System der vorgegebenen Spezifikation entspricht. Wird diese Überprüfung mit Hilfe eines mathematischen Beweises unter Hinzunahme von formalen Methoden durchgeführt, so spricht man von einer „formalen Verifikation“. Voraussetzung ist das Vorliegen einer formalen Beschreibung des Systems und der Anforderungen.

Zwei Verifikationsmethoden haben sich im Laufe der Zeit herauskristallisiert. Einerseits gibt es das „Theorem Proving“. Mathematische Behauptungen, die die Eigenschaften eines Systems betreffen, werden dabei schrittweise über die formale Semantik des Systems hergeleitet. Teilschritte der Herleitung können mit Hilfe von interaktiven Theorembeweisern automatisiert werden. An manchen Stellen ist jedoch der Eingriff des Benutzers notwendig, um bei der Herleitung eine Richtung vorzugeben. Dieser Schritt ist oft unintuitiv und erfordert detaillierte Kenntnisse im Theorem Proving. Folglich ist diese Methode nicht für den alltäglichen Gebrauch durch Entwickler geeignet, die eher mit den Problemen ihrer Anwendungsdomäne beschäftigt sind. Für die Verifikation von sicherheitskritischen Komponenten ist der Aufwand jedoch gerechtfertigt, insofern Verifikationsexperten hinzugezogen werden.

Eine andere Verifikationsmethode stellt das Modelchecking dar, dessen Gründzüge im Folgenden beschrieben werden sollen. Die vorliegende Diplomarbeit konzentriert sich auf diese Methode.

4.2 Modelchecking

Systeme können unterschiedlich ausgeprägt sein. „Transformationelle Systeme“ berechnen eine Funktion. Zunächst werden die Eingabewerte eingelesen, die Berechnung durchgeführt und dann die Ausgabewerte geliefert. Transformationelle Systeme interagieren nur an vordefinierten Stellen mit der Umgebung.

„Reaktive Systeme“ hingegen interagieren ständig mit ihrer Umgebung. Ein reaktives System besteht aus mehreren Komponenten, die parallel zueinander arbeiten. Über den Austausch von Nachrichten kommunizieren sie miteinander und mit der Umgebung. Typischerweise ist ihre Laufzeit unbegrenzt. Unter anderem gehören eingebette Systeme zur Klasse der reaktiven Systeme.

Die, für solche Systeme geltenden Eigenschaften lassen sich klassifizieren. Mit „Sicherheitseigenschaften“ kann man ausdrücken, dass gewisse unerwünschte Zustände nie angenommen werden. „Lebendigkeitseigenschaften“ hingegen drücken aus, dass das System irgendwann schliesslich die gewünschte Eigenschaft haben wird.

Modelchecking ist eine modellbasierte Verifikationstechnik zum Nachweis solcher Eigenschaften für ein formal spezifiziertes System. Im Gegensatz zum interaktiven Theorembeweisen ist Modelchecking eine automatisierte Verifikationstechnik, die während der Überprüfung kein Eingreifen des Benutzers erfordert.

Im Folgenden wird auf Konzepte eingegangen, die zur Anwendung dieser Technik erforderlich sind.

4.2.1 Modellierungssprache

Die Beschreibung des zu verifizierenden reaktiven Systems erfolgt mit Hilfe einer Modellierungssprache. Abhängig vom eingesetzten Werkzeug fällt die Darstellung unterschiedlich aus. Das grundlegende Prinzip ist jedoch allen gemeinsam. Daher wird hier stellvertretend die allgemeine Beschreibungsnotation einer Kripke-Struktur eingeführt mit der das System modelliert werden kann.

Definition 4.2.1 Transitionsgraph

Ein (beschrifteter) Transistionsgraph über Aussagenvariablen p_1, ..., p_n hat die Form

$\mathcal{M} = (S, R, L)$

wobei

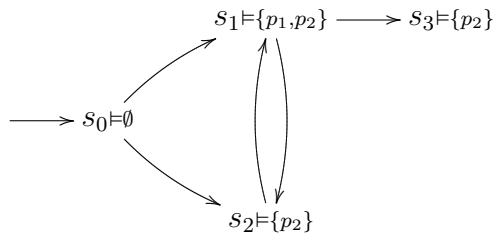
- (S, R) ein gerichteter Graph mit der Knotenmenge S und der Transitionsmenge R ist, und

- L: $S \rightarrow 2^{\{p1,...,pn\}}$ eine Labelfunktion ist. L(s) ist die Menge der p_i, die in s gelten.

Definition 4.2.2 Kripke-Struktur

Das Tupel $(\mathcal{M}, s)$ mit dem Transitionsgraphen $\mathcal{M} = (S, R, L)$ und dem Anfangsknoten $s \in S$ ist eine Kripke-Struktur über p_1, ..., p_n.

Beispiel 4.2.1



In der hier verwendeten Notation bedeutet $s \vDash \{x,y\}$*, dass im Knoten s die Aussagenvariablen x und y erfüllt sind. Der Graph stellt eine Kripke-Struktur* $(\mathcal{M}, s_0)$ *dar, mit* $\mathcal{M} = (S, R, L)$*, wobei*

- $S = \{s_0, s_1, s_2, s_3\}$

- $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_2), (s_1, s_3), (s_2, s_1)\}$

- $L(s_0) = \emptyset,\ L(s_1) = \{p_1, p_2\},\ L(s_2) = \{p_2\},\ L(s_3) = \{p_2\}$

Ausgehend vom Anfangsknoten einer Kripke-Struktur können verschiedene Pfade abgegangen werden. Dies führt zur Definition des Abwicklungsbaumes:

Definition 4.2.3 Abwicklungsbaum

Für eine Kripke-Struktur $(\mathcal{M}, s)$ *über den Aussagenvariablen* $p_1, ..., p_n$ *mit* $\mathcal{M}$ *= (S, R, L) ergibt sich als Abwicklungsbaum wiederum eine Kripke-Struktur:*

$t(\mathcal{M}, s) = (S^*, R^*, L^*)$

mit

- S^* *= Menge der Folgen* $s_0...s_r$ *mit* $s_i \in S,\ s_0 = s$ *und* $(s_i, s_{i+1}) \in R$ *für* $i < r$

- $R^* = \{(s_0...s_r, s_0...s_r s_{r+1}) \in S^* \times S^* \mid (s_r, s_{r+1}) \in R\}$

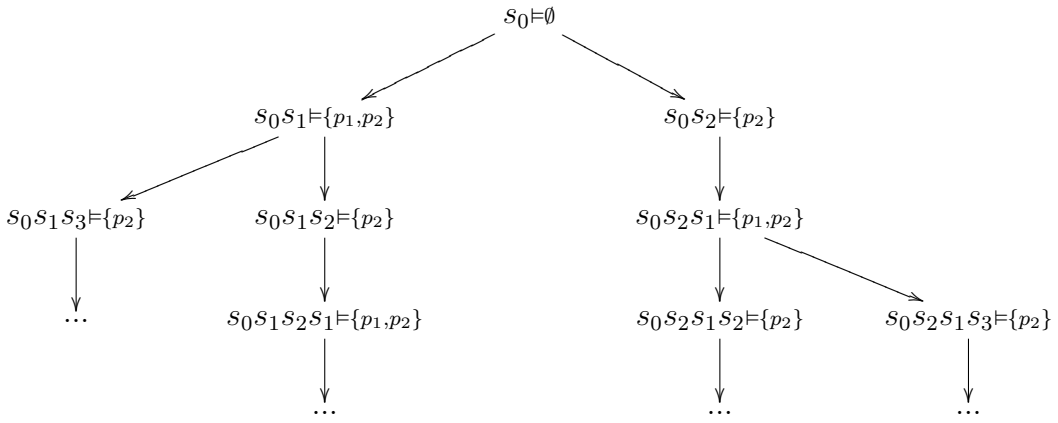
- $L^* :\ S^* \to 2^{p_1,...,p_n}$ *mit* $L^*(s_0...s_r) = L(s_r)$

Beispiel 4.2.2 *Zur Veranschaulichung wird der Abwicklungsbaum* $t(\mathcal{M}, s_0)$ *für das oben angegebene Beispiel* $(\mathcal{M}, s_0)$ *angegeben:*

$S^* = \{s_0,\ s_0s_1,\ s_0s_2,\ s_0s_1s_2,\ s_0s_1s_3,\ s_0s_2s_1,\ s_0s_1s_2s_1,\ s_0s_2s_1s_2,\ s_0s_2s_1s_3,\ s_0s_1s_2s_1s_2, ...\}$

$R^* = \{(s_0, s_0s_1),\ (s_0, s_0s_2),\ (s_0s_1, s_0s_1s_2),\ (s_0s_1, s_0s_1s_3),\ (s_0s_2, s_0s_2s_1),$
$(s_0s_1s_2, s_0s_1s_2s_1), ...\}$

$L^*(s_0) = \emptyset,\quad L^*(s_0s_1) = \{p_1, p_2\},\quad L^*(s_0s_2) = \{p_2\},\quad L^*(s_0s_1s_2) = \{p_2\},$
$L^*(s_0s_1s_3) = \{p_2\},\ L^*(s_0s_2s_1) = \{p_1, p_2\},\ ...$



4.2.2 Spezifikationsprache

Mit Hilfe einer Spezifikationssprache können Eigenschaften, die beim System nachgewiesen werden sollen, formuliert werden. Es existieren verschiedene Varianten, wobei hier nur auf CTL (Computation Tree Logic) und LTL (Linear Time Temporal Logic) eingegangen werden soll. Die Syntax und die Semantik von CTL und LTL finden sich im Anhang A.

Nicht alle Eigenschaften lassen sich mit diesen beiden Sprachen zeigen. Zudem existieren für beide Sprachen Eigenschaften, die in der jeweils anderen Sprache nicht formuliert werden können. Die Ausdrucksmöglichkeiten von CTL und LTL bilden nur eine Schnittmenge.

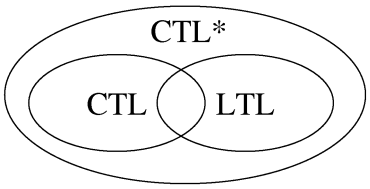


Abbildung 4.1: Inklusionseigenschaft von Spezifikationssprachen.

Da man in LTL Formeln keine Pfadquantoren wie bei CTL verwenden darf, lassen sich LTL Formeln einfacher formulieren. Man muss nicht

immer die Auswirkung auf den gesamten Abwicklungsbaum wie bei CTL im Hinterkopf behalten, sondern betrachtet nur einen Pfad[1].

Weiterführende Informationen können [CGA00] entnommen werden.

4.2.3 Verifikationsalgorithmus

Mit den bisher besprochenen Mitteln ist es nun möglich, ein System in Form einer Kripke-Struktur zu modellieren. Zudem kann dafür mittels einer Spezifikationssprache eine Eigenschaft formuliert werden. Die Eigenschaft liegt somit als CTL- oder LTL-Formel vor, deren Aussagenvariablen den Labeln der Zustände der Kripke-Struktur entsprechen.

Das Ziel des Modelcheckings ist der Beweis der Gültigkeit dieser Eigenschaft für das modellierte System. Für diesen Verifikationsvorgang wird ein algorithmisches Verfahren benötigt. Dieses exploriert die erreichbaren Zustände des Systems und überprüft in jedem Zustand, ob die spezifizierte Eigenschaft erfüllt ist. Dabei wird der Abwicklungsbaum der betrachteten Kripke-Struktur abgegangen[2]. Im Falle der Nichterfüllung ist es möglich, ein Gegenbeispiel in Form eines Pfades durch den Abwicklungsbaum zu generieren. Damit können Fehlerstellen im modellierten System ausgemacht werden.

Nicht immer kann für eine Eigenschaft ein eindeutiges, aussagekräftiges Gegenbeispiel geliefert werden. Ein Beispiel dafür ist die CTL-Formel „EF(p1 $\wedge$ $\neg$p2)“ für die Kripke-Struktur in Beispiel 4.2.1. Hier müssten als Gegenbeispiel alle möglichen Pfade im Abwicklungsbaum aufgezählt werden, die diese Eigenschaft nicht erfüllen, wovon es unendlich viele gibt.

Wie eben angedeutet wurde liegt die Problematik beim Modelchecking in der Anzahl der erreichbaren Zustände. Bei sehr komplexen Systemen steigt diese Anzahl und damit wächst auch der Zustandsraum. Das Problem der „Zustandsraumexplosion“ macht sich in der Praxis bei Erreichen von Speicher- und Zeitlimits schnell bemerkbar. Gängige Modelchecker unterstützen daher unterschiedliche Varianten und Erweiterungen zur Implementierung des Algorithmus, um dieser Hürde entgegenzuwirken. „On-the-fly-Modelchecker“ bauen während des Verifikations-

[1] LTL-Formeln wird implizit ein einziger Pfad-Allquantor vorangestellt.

[2] Dies entspricht dem Vorgehen beim expliziten Modelchecking

vorganges nur den Teil des Zustandsraumes auf, der benötigt wird. Dies reduziert den Speicheraufwand. „Symbolische Modelchecker“ hingegen basieren intern auf einer kompakten Repräsentation des Systems[3], mit dem Ziel, den Zeit- und Speicheraufwand zu reduzieren.

[3]Mittels „Ordered Binary Decision Diagrams“ werden Kripke-Strukturen als boolsche Formeln repräsentiert.

Kapitel 5

Fallbeispiel: Türsteuergerät

Die Bewertung der bisher beschriebenen Qualitätssicherungsmassnahmen erfolgt anhand eines Fallbeispiels aus der Automobilindustrie. Dazu dient eine, im Rahmen eines Projektes am Fraunhofer Institut erstellte Spezifikation [PH02]. In Form eines Lastenheftes sind darin die Funktionen des Steuergerätes im Automobil beschrieben, die das Türmodul und den Fahrzeuginnenraum betreffen. Die Systemkomplexität und Beschreibungstiefe der Spezifikation ist vergleichbar mit denen einfacher Steuergeräte im Fahrzeuginnenraum. Die Kapitel 7 und 8 befassen sich dann mit der Umsetzung dieser textuell verfassten Spezifikation in ein ausführbares Modell. Die Umsetzung erfolgte anhand ausgewählter Komponenten des Türsteuergerätes.

5.1 Einsatz des Türsteuergerätes im Automobil

5.1.1 Funktionen des Türsteuergerätes

Die Spezifikation sieht je Fahrertür ein Türsteuergerät (TSG) vor. Folgendes Bild veranschaulicht die Funktionsgruppen des Steuergerätes und die mit ihnen interagierenden peripheren Komponenten.

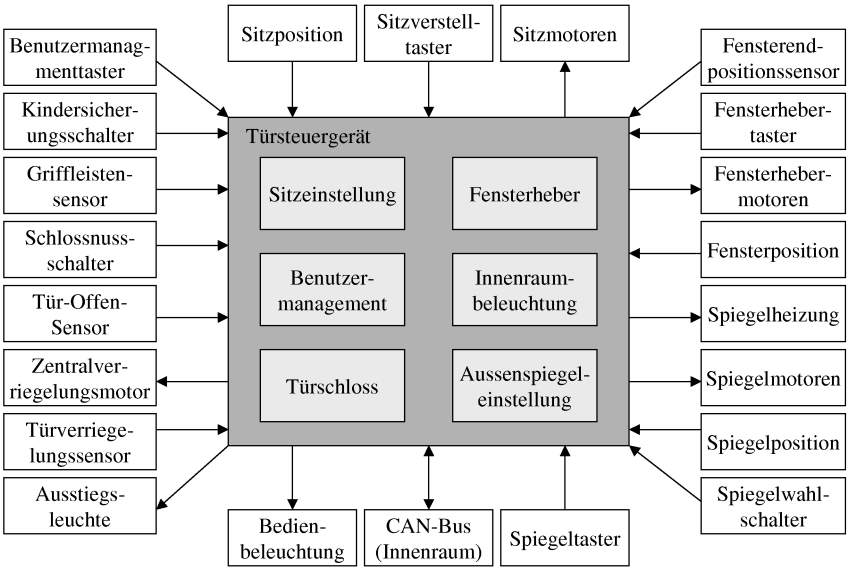


Abbildung 5.1: Funktionen des Türsteuergerätes.

Die Funktionen für die Sitzeinstellung sind zuständig für das Verstellen von Lehnenwinkel, Sitzposition, Sitzhöhe und Schalung des zugeordneten Vordersitzes. Das Speichern von Einstellungen des Fahrersitzes und der Aussenspiegel erfolgt im Benutzermanagement. Für die Aussenspiegeleinstellung ist eine eigene Funktionsgruppe zuständig. Der Funktionalität für das Türschloss, die Fensterheber und die Beleuchtung sind eigene Abschnitte gewidmet. Diese Diplomarbeit beschränkt sich in ihrer Betrachtung auf die drei zuletzt genannten Komponenten.

5.1.2 Hardwareeigenschaften

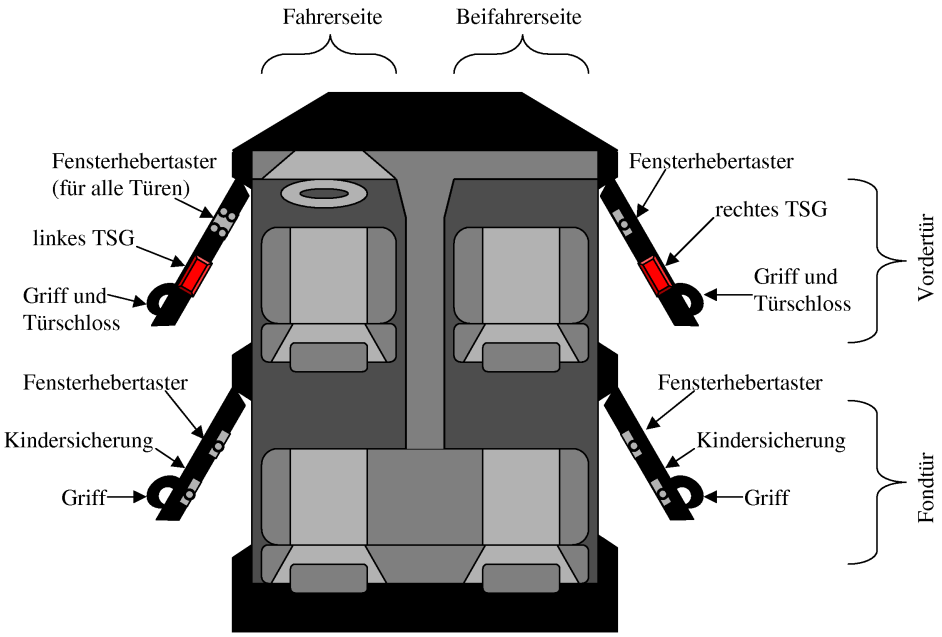


Abbildung 5.2: Position der Türsteuergeräte und Bedienelemente.

Beide Steuergeräte sind an den CAN-Bus angebunden und können somit miteinander und mit anderen Komponenten des Fahrzeugs kommunizieren. Zudem sind je Türsteuergerät auf dessen Fahrzeugseite periphere Komponenten über folgende Schnittstellen angeschlossen: Sensoren und Aktoren, die sich in der Vordertür befinden sind dem Stecker „S1“ zugeordnet. Sensoren und Aktoren auf der gleichen Fahrzeugseite, die nicht in der Vordertür platziert sind, sind über Stecker „S2“ mit dem Steuergerät verbunden. Dazu gehören zum Beispiel Elemente im Aussenspiegel und in der Hintertür. Stecker „S3“ ist für die Anbindung an den CAN-Bus zuständig. Eine detaillierte Zuordnung von, in dieser Diplomarbeit relevanten peripheren Komponenten zu den Steckern ist im Anhang B zu finden.

Im Folgenden wird die grundsätzliche Funktionsweise der, in dieser Diplomarbeit betrachteten Komponenten erläutert. Dies sind das Türschloss, der Fensterheber und die Beleuchtung. Die restlichen Komponenten, sowie die Fehlersicherung und Diagnose sind nicht mehr Gegenstand dieser Arbeit. Details sind [PH02] zu entnehmen.

5.2 Türschloss

Die Türschloss-Funktion ist verantwortlich für die Zentralverriegelung des Fahrzeugs. Betroffen sind alle Türen, wobei bzgl. der Türenanzahl zwischen den Fahrzeugtypen Cabriolet, Coupé und Limousine unterschieden wird. Jedes Türsteuergerät interagiert mit den Aktoren und Sensoren der Tür(en) der eigenen Fahrzeugseite und stimmt sich über den CAN-Bus mit dem anderen Steuergerät ab.

5.2.1 Verriegeln

Ein Verriegeln-Signal über den CAN-Bus oder das Zusperren des Türschlosses an der Vordertür fasst das TSG als Aufforderung zum Verriegeln auf. Die Verriegelung kann allerdings nur dann stattfinden, falls alle Türen geschlossen sind und die Batteriespannung mindestens 9 Volt beträgt. Der Verriegelungsvorgang besteht aus dem Ansteuern der Verriegelungsmotoren und dem Senden einer CAN-Nachricht an das gegenüberliegende TSG mit der Aufforderung zum Verriegeln. Abhängig von der Konfiguration des Steuergerätes erhält auch die Fensterheberkomponente über eine interne Verbindung die Aufforderung zum Schliessen der Fenster. Das Steuergerät kann bei Einbau in ein Cabriolet so konfiguriert werden, dass auch das Verdeck geschlossen wird. Nach mindestens zwei Sekunden werden die Verriegelungsmotoren gestoppt, falls bis dahin die Riegel schon geschlossen wurden. Wenn die Riegel nach drei Sekunden noch immer geöffnet sind, wird der Verriegelungsvorgang abgebrochen und eine Fehlermeldung über den CAN-Bus verschickt. Das Türsteuergerät auf der linken Fahrzeugseite wartet nach erfolgreicher Verriegelung eine Sekunde lang auf eine eventuelle Fehlermeldung des rechten Türsteuergerätes. Nach Ablauf der Sekunde ohne Fehlermeldung wird abschliessend zweimal geblinkt.

5.2.2 Entriegeln

Das mechanische Öffnen einer Tür (z.B. mittels Griff), ein Entriegeln-Signal über den CAN-Bus oder das Aufschliessen des Türschlosses an der Vordertür werden als Aufforderung zum Entriegeln aufgefasst. Auch der inkonsistente Zustand einer geöffneten Tür mit vorgeschobenem Riegel

führt zum Entriegeln. Der Entriegelungsvorgang wird allerdings nur ausgeführt, wenn die Batteriespannung mindestens 9 Volt beträgt. Liegt die Batteriespannung temporär unter 9 Volt, weil der Antriebsmotor gerade gestartet wird, so wird gewartet, bis der Motor gestartet wurde. Liegt die Batteriespannung dann wieder über 9 Volt, so wird der Entriegelungsvorgang wiederholt. Der Entriegelungsvorgang besteht aus dem Ansteuern der angeschlossenen Verriegelungsmotoren. Über das Verschicken einer entsprechenden Nachricht über den CAN-Bus wird das Türsteuergerät auf der gegenüberliegenden Seite ebenso zum Entriegeln aufgefordert. Nach mindestens zwei Sekunden werden die Verriegelungsmotoren gestoppt, falls bis dahin die Riegel schon geöffnet wurden. Wenn die Riegel nach drei Sekunden noch immer geschlossen sind, wird der Entriegelungsvorgang abgebrochen und eine Fehlermeldung über den CAN-Bus verschickt. Das Türsteuergerät auf der linken Fahrzeugseite wartet nach erfolgreicher Verriegelung eine Sekunde lang auf eine eventuelle Fehlermeldung des rechten Türsteuergerätes. Nach Ablauf der Sekunde ohne Fehlermeldung wird abschliessend einmal geblinkt.

Prinzipiell sollen sich alle Türriegel nach Beendigung der Aktionen im gleichen Zustand befinden. Dies setzt eine Synchronisation beider Türsteuergeräte über den CAN-Bus voraus. Ein Konflikt bei gleichzeitiger Aufforderung zum Ver- und Entriegeln wird durch eine höhere Priorität der Entriegelung gelöst.

5.3 Fensterheber

Die Systemfunktion des Fensterhebers ist für das Öffnen und Schliessen der Fenster zuständig. Abhängig vom Fahrzeugtyp (Limousine, Cabriolet oder Coupé) und der Position des Türsteuergerätes (linke bzw. rechte Fahrzeugseite) reagiert die Fensterheberkomponente auf Anweisungen[1] von bestimmten Eingangsquellen. Quellen können der CAN-Bus, die vorderen bzw. hinteren Fensterhebertasten und das Türschloss sein. Die Aufforderungen werden entweder an das gegenüberliegende Türsteuergerät weitergeleitet oder selbst weiterverarbeitet, abhängig davon, ob sich die betroffene Fensterscheibe auf derselben oder gegenüberliegenden Fahrzeugseite wie das TSG befindet.

[1] Mit einer Anweisung ist die Aufforderung zum Öffnen bzw. Schliessen einer bestimmten Fensterscheibe gemeint.

5.3.1 Fensteröffnen bzw. -schliessen

Sind die eingehenden Aufforderungen für dieselbe Fahrzeugseite, wie die des Türsteuergerätes bestimmt, so wird zunächst die Batteriespannung auf einen Wert von mindestens 10 Volt überprüft. Ist diese Voraussetzung erfüllt, wird der Fensterhebermotor entsprechend angesteuert.

Folgende Bedingungen führen dazu, dass der Fensterhebermotor gestoppt wird:

- Loslassen der Fensterhebertaste bei manueller Ansteuerung[2]
- Eintreffen eines anderen Befehls
- Erreichen des Anschlags am Fensterrand
- Erreichen des Timeouts von drei Sekunden
- Stoppen der Scheibe, obwohl der Anschlag am Fensterrand noch nicht erreicht worden ist

Der letzte Fall führt beim Fensterschliessen zur Aktivierung des Einklemmschutzes. Dabei wird das Fenster geöffnet, ohne auf weitere Aufforderungen des Benutzers zu reagieren.

5.4 Beleuchtung

Innenraumbeleuchtung:

Grundsätzlich erlischt die Innenraumbeleuchtung, sobald die Motorzündung aktiviert wird. Ansonsten wird das Licht für 10 Minuten eingeschaltet, wenn eine Tür geöffnet wurde. Wurden alle Türen geschlossen, so leuchtet das Licht noch für 30 Sekunden. Die Ansteuerung

[2]Das „Überdrücken“ der Fensterhebertaste ermöglicht automatisches Öffnen bzw. Schliessen des Fensters bis zum Anschlag. Die Taste muss dann nicht weiter gedrückt gehalten werden, wie beim manuellen Öffnen bzw. Schliessen. Auf Grund von technischen Einschränkungen kann die Taste vom automatischen in den neutralen Zustand nur über das Passieren des manuellen Zustands wechseln. Damit der automatische Fensterlauf dabei nicht abgebrochen wird, soll ein direkter Wechsel vom automatischen in den manuellen Tastenzustand erst nach einer Verzögerung von 0,5 Sekunden erkannt werden.

per Hand und die Überwachung der Batteriespannung wird vom Deckensteuergerät übernommen.

Ausstiegsleuchten:

Die Ausstiegsleuchte einer geöffneten Tür wird bei entsprechender Batteriespannung eingeschaltet.

Kapitel 6

Eingesetzte Werkzeuge

Wie in Abschnitt 2.2 angedeutet, wird der modellbasierte Entwicklungsprozess durch den Einsatz von CASE-Werkzeugen unterstützt. Solche Werkzeuge kamen auch bei der Umsetzung der Spezifikation des Türsteuergerätes zur Anwendung. Im Folgenden werden daher die, in dieser Diplomarbeit verwendeten Tools vorgestellt.

6.1 ASCET-Werkzeugkette

Die ETAS Group [Gmb] richtet sich mit ihren Produkten und Dienstleistungen an die OEMs und Zulieferer aus der Automobilbranche. Dazu stellt sie unter anderem für die Entwicklung von Steuergerätesoftware eine Werkzeugkette zur Verfügung, die den Entwicklungsprozess durchgängig unterstützen soll. In Anlehnung an das V-Modell zeigt Bild 6.1 welche Aufgabengebiete dabei abgedeckt werden.

Die ASCET-Softwarefamilie adressiert hierbei die Schritte vom Entwurf bis zur Implementierung. Sie ist gegliedert in ASCET-MD (Modeling & Design), ASCET-RP (Rapid-Prototyping) und ASCET-SE (Software Engineering).

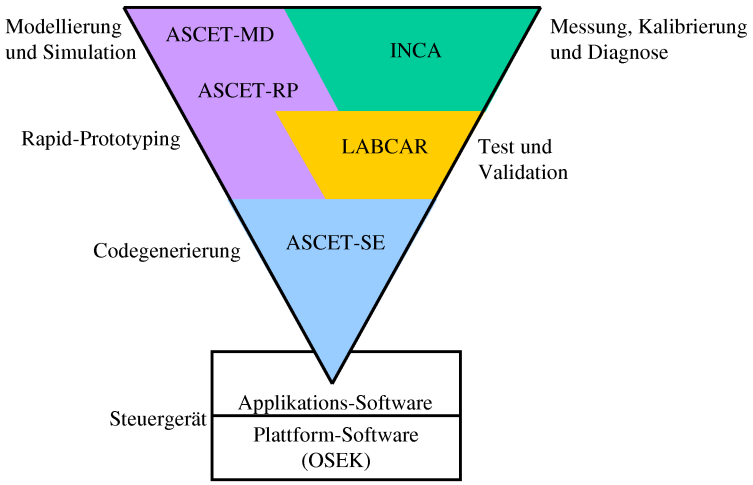


Abbildung 6.1: ASCET-Toolkette.

6.1.1 ASCET-MD

ASCET-MD stellt eine Entwicklungsumgebung zur Verfügung, in der Systemmodelle erstellt werden können. Bild 6.2 veranschaulicht die Struktur eines Projektes in ASCET-MD. Darin wird die Integration der Applikation in das zugrunde liegende OSEK-Betriebssystem deutlich.
Einem Projekt können OSEK-Tasks zugewiesen werden. Entsprechend den Erläuterungen zu OSEK in Abschnitt 2.3 ist jeder Task mindestens ein Prozess zugeordnet. Die Reihenfolge der Task- und Prozessausführungen kann über das Scheduling von OSEK in ASCET konfiguriert werden. In ASCET-MD werden Prozesse innerhalb von Modulen definiert. Module bilden damit die Komponenten auf der höchsten Ebene eines hierarchisch strukturierten Projektes. Die Kommunikation zwischen mehreren Modulen wird durch den Austausch von Nachrichten modelliert. Die Prozesse eines Moduls können auf dessen Nachrichteneingänge und -ausgänge zugreifen und darauf basierend Berechnungen durchführen. Dazu können gemischt graphische und textuelle Elemente für Berechnungen eingesetzt werden. Als textuelles Beschreibungselement dient die Java-ähnliche Beschreibungssprache ESDL (Embedded Software Description Language). ESDL verzichtet dabei auf Vererbung, implementierungsabhängige Konzepte (z.B. Zeiger) und beschränkt sich auf die, im Umfeld von eingebetteten Systemen relavanten Konzepte. Diese Einschränkungen erlauben eine semantische Überprüfung auf

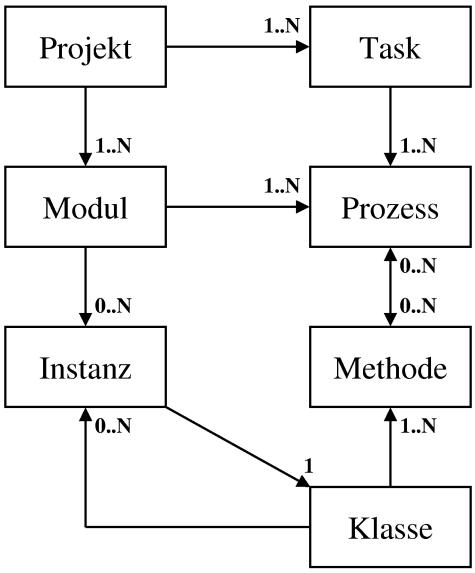


Abbildung 6.2: Projektstruktur.

Korrektheit. Auch C-Code kann für hardwarespezifische Ergänzungen eingefügt werden. Blockdiagramme und Zustandsautomaten stellen die graphischen Elemente dar. Bild 6.3 illustriert anhand eines Beispiels die Definition von Prozessen. Das darin gezeigte Modul definiert ein Integrier-Glied mit zwei Prozessen „init“ und „calc“. Die, im Blockdiagramm den Prozessenamen vorangestellten Ziffern legen den Berechnungsablauf je Prozess fest. „calc“ führt zwei Schritte hintereinander aus. Bei jedem Aufruf von „calc“ durch den Scheduler wird die, seit dem letzten Aufruf vergangene Zeit „dT“ mit dem Eingangssignal „u“ und einem Parameter „K“ multipliziert und zum bisherigen Wert von „y“ hinzuaddiert. Der zweite Schritt besteht in der Ausgabe des aktuell integrierten Wertes. Der Prozess „init“ besteht nur aus einem Initialisierungsschritt.
Bild 6.4 zeigt die Zuweisung der Prozesse zu den OSEK-Tasks.

Oft wiederverwendete und hierarchisch untergliederte Komponenten in Modulen können mit Hilfe von Klassen deklariert und strukturiert werden. Klassen können mit den selben Beschreibungselementen der Module erstellt werden. Allerdings werden Berechnungsabläufe in Klassen nicht in Prozessen, sondern in Methoden mit Ein- und Ausgabeparametern zusammengefasst. Die Definition von Methoden erfolgt analog zum Vor-

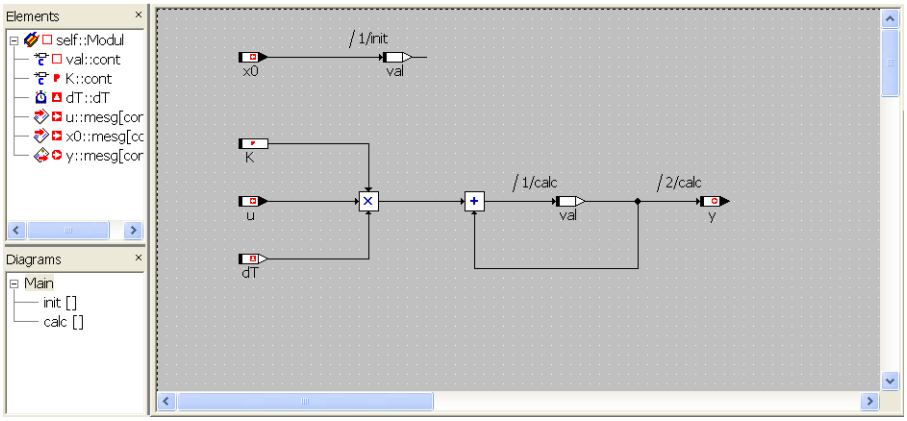


Abbildung 6.3: Definition von Prozessen.

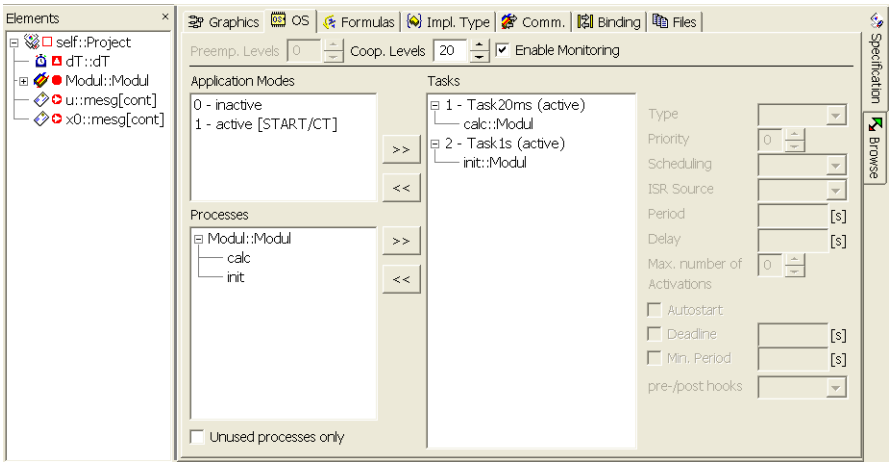


Abbildung 6.4: Scheduling in ASCET.

gehen bei Modulen.

Eine Klasse stellt eine Art Template dar. Dessen Instanzen können in Modulen mehrfach eingesetzt werden. Die Prozesse in diesen Modulen können somit auf die Methoden der Klasse zurückgreifen. Der Aufruf einer Methode stösst den dazugehörigen Berechnungsablauf an und liefert ggf. ein Ergebnis über den Ausgabeparameter zurück. Oft wiederverwendete Berechnungen und Strukturen lassen sich damit in Bibliotheken zusammenfassen.

Mit diesen Modellierungskonstrukten lassen sich diskrete sowie konti-

nuierliche regelungstechnische Berechnungsabläufe darstellen. Aus den erstellten Modellen kann in ASCET-MD Code für die Entwicklungsplattform generiert werden. Sie lassen sich somit simulieren. Details können [Gmb03a, Gmb03b] entnommen werden.

6.1.2 ASCET-RP

Rapid-Prototyping mit Hilfe von ASCET-RP ermöglicht die Validierung eines ASCET-MD-Modells. Dafür bietet ETAS Hardwarelösungen an, die sich in ein Fahrzeug einbauen lassen und mit dessen Aktoren bzw. Sensoren interagieren können. Beim Rapid-Prototyping wird typischerweise solch ein Gerät anstelle des eigentlichen Steuergerätes im Regelkreis platziert. Für das, in ASCET-MD erstellte Modell wird anschliessend mit der ASCET-RP-Erweiterung der entsprechende Code für die Prototyping-Hardware generiert und auf das Gerät geladen. Damit lässt sich das Modell in der realen Umgebung ausführen. Die Funktionalität kann damit schon frühzeitig während des Entwicklungsprozesses validiert werden.

6.1.3 ASCET-SE

Der Schwerpunkt von ASCET-MD bzw. ASCET-RP liegt in der Funktionsentwicklung. Im Gegensatz dazu konzentriert sich ASCET-SE auf die Codegenerierung. Bei der Codegenerierung für die Zielplattform sind eingeschränktere Ressourcen zu beachten. Dazu gehören zum Beispiel der begrenzte Speicherplatz und die beschränkten Datentypen der Zielplattform. Ebenso kann eine Abbildung der Gleitpunktarithmetik in physikalischen Gleichungen auf die Fixpunktarithmetik vorgenommen werden. ASCET-SE kapselt die Unterschiede zwischen verschiedenen Plattformen, so dass ASCET-MD damit auf eine einheitliche Schnittstelle zurückgreifen kann. Erst bei der Codegenerierung im Rahmen von ASCET-SE werden target-spezifische Aspekte berücksichtigt.

6.2 Unterstützende Werkzeuge

Die folgenden Abschnitte beschreiben die Werkzeuge, die im Zusammenhang mit ASCET und den Testmethoden aus Abschnitt 7.3 Anwendung fanden.

6.2.1 IBM Rational Test RealTime

Zur Überdeckungsanalyse bei der Anwendung von konventionellen Qualitätssicherungsmassnahmen wurde auf das Testframework „Test RealTime“ von IBM Rational zurückgegriffen. Damit lässt sich C-Code instrumentieren (siehe Abschnitt 3.2.3), um Überdeckungsanalysen wie die C_0- oder MC/DC-Überdeckung durchzuführen. Das Resultat eines Testlaufs lässt sich rückwirkend farbig im Quelltext darstellen. Grün eingefärbte Zeilen haben die Anforderung des ausgewählten Überdeckungskriteriums während des Tests erfüllt, rote hingegen nicht.

6.2.2 CTE XL

Ein weiteres Werkzeug, das im Zusammenhang mit konventionellen Testverfahren eingesetzt wird, ist CTE XL. Damit lassen sich Testfälle grafisch erstellen und festhalten. In Form einer Matrix können, abhängig vom Zeitpunkt Werte für jeden Eingabeparameter des Testobjekts angegeben werden. Diese Testfälle lassen sich dann exportieren und in ein entsprechendes Format für die Eingabe in das Testobjekt wandeln.

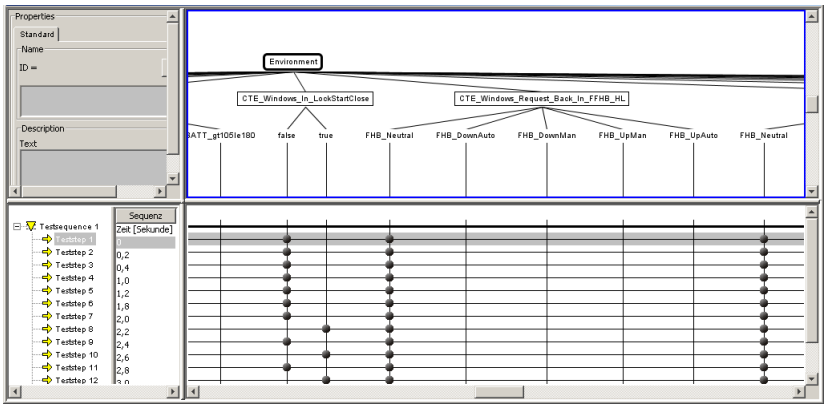


Abbildung 6.5: CTE/XL Screenshot.

6.3 AutoFocus

AutoFocus ist ein frei erhältliches Systementwicklungswerkzeug, das an der Technischen Universität München entwickelt wurde. Es basiert auf Konzepten der formalen Beschreibungsmethodik FOCUS [BS01].

FOCUS dient zur formalen Spezifikation und Entwicklung verteilter, reaktiver Systeme mit Hilfe von präzise definierten, formalen Darstellungsmitteln. Zum Einsatz kommen dabei intuitive entwicklungsorientierte Beschreibungstechniken. Ein verteiltes System wird dabei als Verbund von Komponenten modelliert. Über Kanäle sind diese miteinander verbunden und tauschen darüber Nachrichten aus.

In AutoFocus findet sich ein Teil dieser Konzepte in Form von graphischen Beschreibungsmitteln wieder [BHS99]. Dem gesamten System in AutoFocus liegt ein globaler periodischer Systemtakt zu Grunde. Je Systemtick führen alle Komponenten gleichzeitig jeweils einen Berechnungsschritt aus. Nachrichten können somit bei jedem Systemtick empfangen und gesendet werden.

Zur Modellierung eines Systems bietet AutoFocus mehrere Diagrammarten an. Folgende Abschnitte gehen darauf ein.

6.3.1 System Structure Diagram (SSD)

Ein Systemstrukturdiagramm (SSD) beschreibt den inneren Aufbau einer Komponente. Diese kann aus mehreren Unterkomponenten bestehen, die im SSD in Form von Blöcken dargestellt werden. Nachrichten können über Kanäle zwischen den einzelnen Komponenten ausgetauscht werden. Dazu dienen Verbindungslinien zwischen den einzelnen Blöcken, die über Ports angebunden werden. Unterkomponenten können wiederum rekursiv in weitere Unterkomponenten zerlegt werden. Für jede hierarchisch untergliederte Komponente existiert ein eigenes SSD.

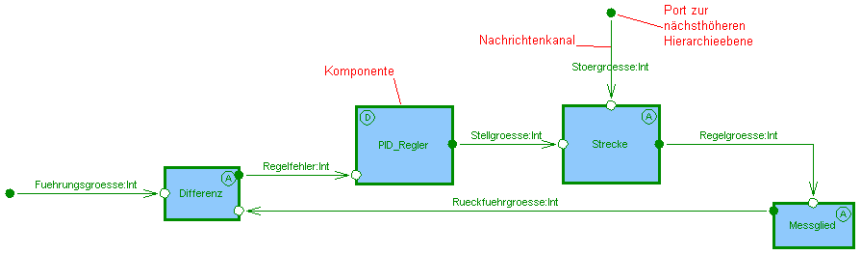


Abbildung 6.6: Beispiel für ein SSD.

6.3.2 State Transition Diagram (STD)

Jeder nicht weiter hierarchisch untergliederten Komponente in einem SSD kann ein Zustandsübergangsdiagramm (STD) zugeordnet werden. Ein STD spezifiziert das Verhalten einer solchen atomaren Komponente. Es besteht aus Zuständen in Form von Ovalen. Beschriftete Verbindungen zwischen diesen stellen Transitionen dar. Eine Komponente kann sich zu jedem Zeitpunkt in nur einem Zustand befinden. Der Übergang in einen anderen Zustand erfolgt über eine der anliegenden Transitionsverbindungen. Dazu werden bei jedem Systemtick die Transitionsbeschriftungen, bestehend aus Priorität, Vorbedingung, Eingabe, Ausgabe und internen Aktionen ausgewertet.

Beim Eintreffen des Systemticks werden in jeder STD-Komponente folgende Schritte durchgeführt: Die aktuell anliegenden Werte an den Nachrichteneingängen werden zunächst ausgelesen und an Variablen gebunden. Damit werden dann die prädikatenlogischen Vorbedingungen ausgewertet. Eine Transition kann nur dann schalten, falls ihre Vorbedingung

erfüllt ist. Von diesen aktiven Transitionen werden nur diejenigen mit der höchsten Priorität betrachtet. Falls mehrere Transitionen gleichzeitig schalten können, wird von diesen nichtdeterministisch eine Transition ausgewählt. Das Schalten dieser Transition geschieht mit der Aktualisierung von lokalen Variablen und der Ausgabe von Nachrichten. Diese Nachrichten stehen dann für andere Komponenten beim nächsten Systemtick zur Verfügung.

Das zugehörige SSD kann statt der normalen Ports auch über sogenannte „Immediate Ports“ an die anderen Komponenten angebunden sein. Falls dies der Fall ist, stehen die Nachrichten schon im selben Systemtick den anderen Komponenten zum Auslesen zur Verfügung.

Der Ablauf ist formal durch die Semantik von AutoFocus definiert und kann [Wim05] entnommen werden.

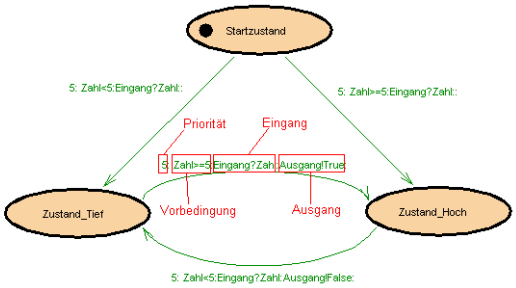


Abbildung 6.7: Beispiel für ein STD.

6.3.3 Extended Event Traces (EET)

Zur Beschreibung von konkreten Systemabläufen werden Ereignisdiagramme eingesetzt. Der zeitliche Verlauf des Nachrichtenaustausches zwischen mehreren Komponenten kann damit aufgezeichnet werden. EETs können zur Darstellung von Gegenbeispielen beim Modelchecking verwendet werden.

6.3.4 Data Type Definitions (DTD)

Benutzerdefinierte Datentypen können für lokale STD-Variablen oder für die zu übertragenden Nachrichten verwendet werden. Sie befinden sich mit den, auf ihnen operierenden Funktionen in Diagrammen für Datentypdefinitionen (DTD).

Die vier erwähnten Diagrammarten stellen jeweils eine eigene Sicht auf ein konkretes modelliertes System dar [SPHP02]. Diesen Beschreibungstechniken liegt als gemeinsamer Nenner ein Metamodell zu Grunde, das die Beschreibungselemente der Diagrammarten miteinander verbindet.

Mit AutoFocus wird eine Plattform für die Integration von formalen Methoden und zur Anbindung an externe Werkzeuge zur Verfügung gestellt. Ein Beispiel dafür wird im nächsten Abschnitt näher erläutert.

6.3.5 Modelchecking-Anbindung

AutoFocus unterstützt unter anderem die Anbindung von Verifikationswerkzeugen, wie dem symbolischen Modelchecker SMV [Cad]. Damit lassen sich Eigenschaften von modellierten Systemen nachweisen. Das Haupteinsatzgebiet des Modelcheckers SMV sind synchrone Systeme. Auf Grund ihres globalen Systemticks sind AutoFocus-Systeme gut dafür geeignet.

Die zu überprüfenden AutoFocus-Modelle müssen zunächst in ein, für den SMV-Modelchecker lesbares Format gewandelt werden, das einer codierten Kripke-Struktur entspricht (siehe Bild 6.8). Diesen Schritt übernimmt die Modelchecking-Anbindung. Das detaillierte Übersetzungsschema kann [Wim00] entnommen werden. Die STDs der atomaren Komponenten werden dabei zunächst jeweils in Formeln aussagenlogischer Art gewandelt. Je STD legt die resultierende Formel einerseits den Initialzustand von lokalen Variablen und Nachrichtenausgängen fest. Andererseits sind darin alle möglichen Zustandsübergänge codiert.

Bislang unterstützte die Modelchecking-Anbindung bei der Übersetzung von Transitionen keine Prioritäten. In SMV wurden somit Transitionen trotz unterschiedlicher Priorität als gleichberechtigt angesehen. Im Rahmen der Diplomarbeit wurde die Modelchecking-Anbindung um die

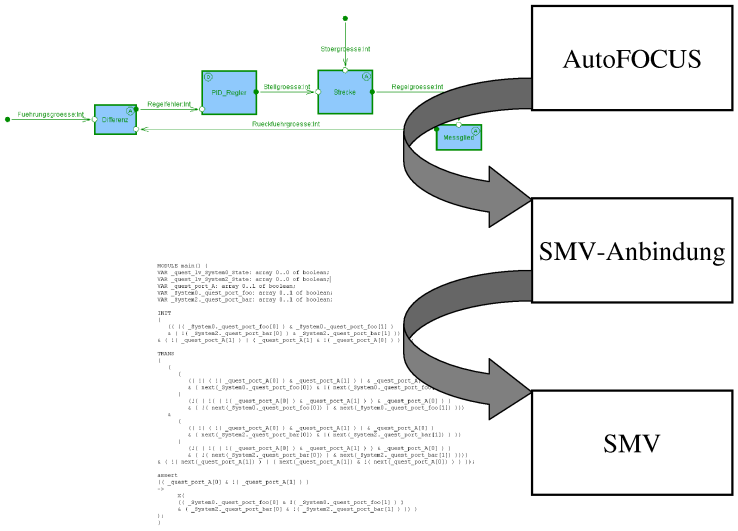


Abbildung 6.8: Modelchecking in AutoFocus.

Unterstützung von Transitionsprioritäten erweitert. Dies soll im folgenden erläutert werden.

Gemäss Abschnitt 6.3.2 erfolgt die Auswertung der Priorität einer Transition bei der Wahl der zunächst schaltenden aktiven Transition. Eine Transition wird als aktiv eingestuft, falls ihre Vorbedingung erfüllt ist. Dies ist der Fall, wenn die, dieser Transition zugehörige übersetzte prädikatenlogische Formel im aktuellen Systemzustand als wahr ausgewertet wird. Ausschlaggebend ist der Teil der Formel, der die Vorbedingung der Transition repräsentiert. Die Erweiterung um Prioritäten entspricht dem Hinzufügen einer weiteren Vorbedingung für diese Transition: Die Nichterfüllung der Vorbedingungen aller Transitionen höherer Priorität. Damit ist sichergestellt, dass Transitionen niedrigerer Priorität nicht schalten können, wenn höherpriorisierte Transitionen aktiv sind.

Anhand eines Beispiels wird dieser Vorgang veranschaulicht. Das zum Beispiel-SSD in Bild 6.9 gehörige STD ist in Bild 6.10 dargestellt.

Ohne Unterstützung von Prioritäten würde die resultierende Formel für die Beschreibung der drei Transitionsübergänge informell folgende Form annehmen:

(State=“Initial“)∧(next(State)=“Two“)∧(Input=“2“)∧(next(Output)=“True“)
∨
(State=“Initial“)∧(next(State)=“One“)∧(Input=“1“)∧(next(Output)=“True“)

Abbildung 6.9: SSD zur Demonstation der Modelchecking-Anbindung

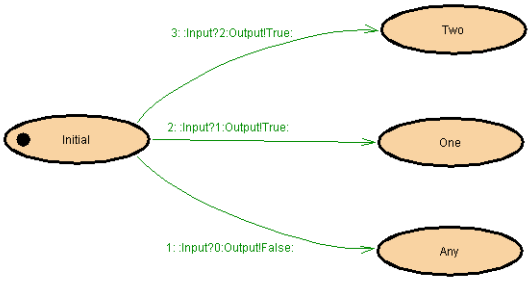


Abbildung 6.10: STD zur Demonstration der Modelchecking-Anbindung

$$\vee$$
$$(State=\text{“Initial“})\wedge(next(State)=\text{“Any“})\wedge(Input=\text{“0“})\wedge(next(Output)=\text{“False“})$$

„State“ bezeichnet hierbei den aktuellen Zustand, „next(State)“ den Nachfolgezustand, „Input“ den aktuell anliegenden Eingabewert und „next(Output)“ die Ausgabe beim nächsten Systemtick.

Die Berücksichtigung von Prioritäten führt auf folgende Formel:

$$(State=\text{“Initial“})\wedge(next(State)=\text{“Two“})\wedge(Input=\text{“2“})\wedge(next(Output)=\text{“True“})$$
$$\vee$$
$$(State=\text{“Initial“})\wedge(next(State)=\text{“One“})\wedge(Input=\text{“1“})\wedge(next(Output)=\text{“True“})\wedge\neg(Input!=\text{“2“})$$
$$\vee$$
$$(State=\text{“Initial“})\wedge(next(State)=\text{“Any“})\wedge(Input=\text{“0“})\wedge(next(Output)=\text{“False“})\wedge$$
$$\neg((Input!=\text{“1“})\vee(Input!=\text{“2“}))$$

Die, aus allen STDs resultierenden Formeln werden im nächsten Schritt zu einer einzigen zusammengefasst. Diese repräsentiert alle möglichen, parallel ausführbaren Berechnungsschritte des Gesamtsystems. Anschliessend wird sie so umcodiert, dass benutzerdefinierte Datentypen und Funktionen eliminiert werden und letztendlich eine Boolsche Formel vorliegt. Der SMV-Modelchecker ist nun im Stande, das Modell in Form dieses Ausdrucks einzulesen und damit die spezifizierte Eigenschaft zu überprüfen. Das Resultat und gegebenenfalls ein Gegenbeispiel werden wieder in das AutoFocus-Framework importiert und können dort in Form von EETs visualisiert werden.

Kapitel 7

Modellierung in ASCET

Dieses Kapitel besteht aus zwei wesentlichen Teilen. Der erste Teil befasst sich mit der Modellierung der Türsteuergeräte-Software (siehe Kapitel 5) in ASCET (siehe Abschnitt 6.1). Die nachfolgenden Abschnitte beschreiben die Anwendung von konventionellen Qualitätssicherungsmassnahmen auf das Resultat (vgl. Kapitel 3).

7.1 Modell des Türsteuergerätes

7.1.1 Rahmenbedingungen

Ein Modell stellt im Allgemeinen die Abstraktion eines realen Zusammenhangs dar. Ein Grund dafür ist die Konzentration auf das Wesentliche. Technische Details sollen damit den Entwicklungsprozess nicht vernebeln. Für die technische Realisierung eines solchen Modells sind letztenendes konkretisierende Massnahmen erforderlich. Im vorliegenden Fall ist dies zum Beispiel die automatische Codegenerierung.

Voraussetzung für einen reibungslosen Übergang vom Modell zur Realisierung sind definierte Schnittstellen und Rahmenbedingungen, die an die Umgebung des Steuergeräts gestellt werden.

Schnittstellenkonvention

Die Stecker S1, S2 und S3 stellen die Schnittstelle des Steuergerätes zur Umgebung dar (siehe Anhang B). Es werden Signale über Steckerpins (S1, S2) und den CAN-Bus (S3) übertragen. Die Modellierung dieser externen Schnittstellen in ASCET ist in dieser Diplomarbeit durch folgendes Schema festgelegt worden:

Externe Signale werden mit Hilfe von ASCET-Nachrichten[1] modelliert. Abhängig von der Art des Signals, können der zugehörigen Nachricht verschiedene Werte zugewiesen werden. Dies wird in ASCET mit Hilfe von Datentypen modelliert. Dazu stehen Standard-Datentypen (z.B. Bool, Int) zur Verfügung. Alternativ können benutzerdefinierte Datentypen definiert werden. Anhang C listet die verwendeten benutzerdefinierten Datentypen und die darauf basierenden ASCET-Nachrichten auf. Konkreten Signalpegeln auf S1 und S2 wird somit eine Bedeutung zugewiesen. Dasselbe gilt für die CAN-Nachrichten, die über S3 übertragen werden.

Auch das zeitliche Verhalten der Signalübertragung muss festgelegt werden. Dieses richtet sich nach der Art des Signalempfängers bzw. -senders und unterliegt der Konvention in Tabelle 7.1.

[1] Im Folgenden bezieht sich der alleinstehende Begriff „Nachricht“ auf Nachrichten in ASCET und nicht auf CAN-Nachrichten.

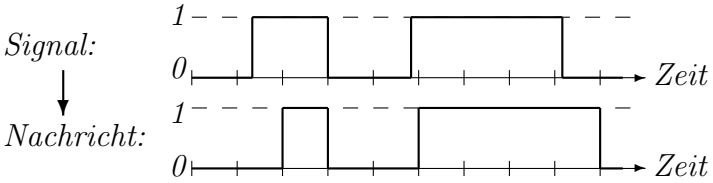
	Eingänge	Ausgänge
Signale über S1, S2	Sensoren liefern ständig ein Signal an das TSG; Nachrichtenwert entspricht immer dem aktuell gelesenen Sensorwert	Aktuatoren erhalten ständig ein Stellsignal vom TSG; Nachrichtenwert entspricht immer[2] dem aktuellen Stellwert
periodische Signale über CAN-Treiber	über den CAN-Bus wird an periodischen Zeitpunkten ein Signal empfangen; Nachrichtenwert entspricht dem zuletzt übertragenen Signalwert	periodisch zu sendendes Signal wird dem CAN-Treiber übermittelt; Nachrichtenwert entspricht dem periodisch zu übertragenden Signalwert
spontane Signale über CAN-Treiber	ein spontanes Signal wird in Form eines Impulses vom CAN-Treiber weitergereicht; Nachrichtenwert wird als Impuls dem Modul übermittelt	ein spontanes Signal wird als Impuls dem CAN-Treiber übermittelt; Nachrichtenwert wird vom Modul in Impulsform ausgegeben[3]

Tabelle 7.1: Schnittstellenkonvention in ASCET.

Die Anbindung an den CAN-Treiber ist kontextspezifisch. Daher wurde in dieser Arbeit diese generische Schnittstelle gewählt. Die Einbettung in eine konkrete Umgebung erfolgt dann mit Hilfe von separaten Wrapper-Modulen, die die Ein- und Ausgabe den Bedürfnissen entsprechend anpassen.

Beispiele sollen die eben eingeführte Schnittstellenkonvention verdeutlichen.

Beispiel 7.1.1 *Auslesen eines Sensorsignals:*



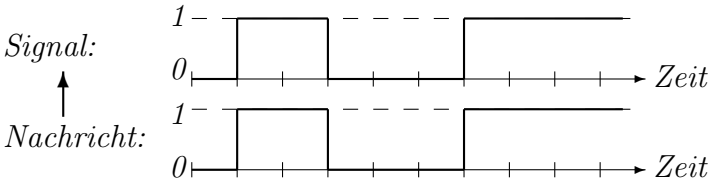
Auf den Zeitachsen sind in periodischen Abständen die Zeitpunkte ge-

[2]Das Anliegen eines kontinuierlichen Signals wird letztenendes durch die Hardware realisiert. Denn der Prozess, der den aktuellen Ausgabewert berechnet, ist bedingt durch das Task-Scheduling nicht andauernd aktiv.

[3] Ein Impuls wird in ASCET erzeugt, indem eine Nachricht in einem Prozess auf den zu übertragenden Wert gesetzt wird. Derselbe Prozess setzt diese Nachricht bei seinem nächsten Aufruf wieder zurück. Somit entsteht ein Impuls der Länge einer Schedulingrunde.

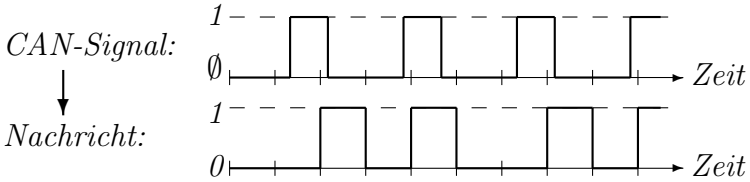
kennzeichnet, an denen der bearbeitende Prozess vom Dispatcher aufgerufen wird. Der Pfeil zwischen den Beschriftungen verdeutlicht die Abhängigkeit des ASCET-Nachrichtenwertes vom aktuellen Signalwert. Die Nachrichtenwerte 0 und 1 werden als Sensorwerte interpretiert.

Beispiel 7.1.2 *Ausgeben eines Stellwertes:*



Hier hängt die Signalausgabe von der ASCET-Nachricht ab. Die Nachrichtenwerte 0 und 1 werden als Stellwerte interpretiert.

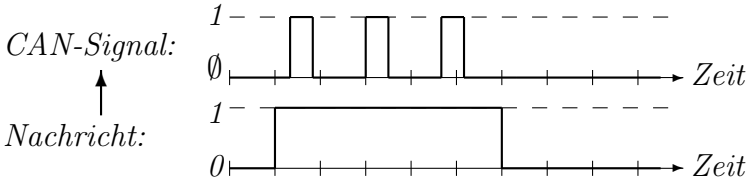
Beispiel 7.1.3 *Empfangen eines periodischen CAN-Signals:*



Nur der Nachrichtenwert 1 wird als gültiges CAN-Signal interpretiert. 0 hingegen wird hier verwendet, um auszudrücken, dass nichts empfangen wurde. Zu CAN-Signalen muss in ASCET somit als Workaround ein separater Wert definiert werden, um eine leere Nachricht auszudrücken. In AutoFocus hingegen sind per Definition leere Nachrichtenkanäle erlaubt.

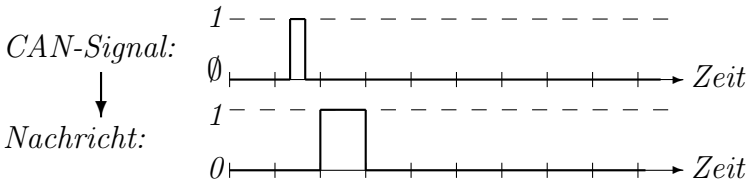
Das Puffern von eingetroffenen Signalen erfolgt im CAN-Treiber. Hier wird nur die Schnittstelle definiert, auf der das Steuergerät aufsetzt.

Beispiel 7.1.4 *Periodisches Versenden eines CAN-Signals:*

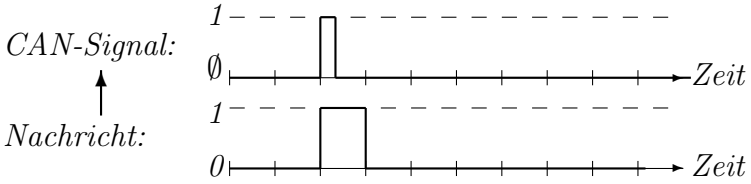


Wie in Beispiel 7.1.3 wird hier 0 verwendet, um auszudrücken, dass nichts gesendet werden soll.

Beispiel 7.1.5 *Empfangen eines spontanen CAN-Signals:*



Beispiel 7.1.6 *Spontanes Versenden eines CAN-Signals:*



Eine weitere Schnittstellenfestlegung betrifft die Abstraktion der Signale, die die Gleichstrommotoren ansteuern. An beide Anschlüsse eines Motors (hier z.B. für die Verriegelung) kann über

FZENTR_MOTOR1:=True;

FZENTR_MOTOR2:=False;

eine Spannung angelegt werden. Eine Negation beider Ausdrücke vertauscht die Polung. Um das Modell übersichtlicher zu halten wurden beide Signale zu einer ASCET-Nachricht zusammengefasst:

FZENTR_MOTOR:=FZENTR_MOTOR_Open

bzw.

FZENTR_MOTOR:=FZENTR_MOTOR_Close

Annahmen an die Umgebung

Die Systemumgebung muss sicherstellen, dass obige Schnittstellenkonventionen eingehalten werden. Dies kann softwaremässig durch die Anpassung mit Hilfe von Wrapper-Modulen geschehen. Andernfalls basieren die Berechnungen ggf. auf falschen Werten, was zu nicht vorhersehbaren Folgen führen kann (z.B. beim Einklemmschutz). Die Hardware muss dafür sorgen, dass die anliegenden Sensorsignale einen definierten Zustand einnehmen (z.B. durch die Entprellung von Tastern). Die Messung von, sich kontinuierlich ändernden Werten (z.B. Batteriespannung) sollte

mit einer Hysterese[4] versehen sein.

Ebenso ist bei Systemstart eine korrekte Vorinitialisierung der ASCET-Nachrichten für einen reibungslosen Ablauf notwendig. Insbesondere müssen erst die CAN-Nachrichten ausgelesen werden, wie z.B. „RL“, um entscheiden zu können, ob das Fahrzeug ein Rechts- oder Linkslenker ist (siehe TSG-Spezifikation). Dies erfordert eine Wartezeit von mindestens 500ms beim ersten Start, da dies das grösste Zeitintervall ist, mit dem periodische CAN-Nachrichten auftreten. Die Werte können z.B. im NVRAM abgespeichert werden, so dass diese Wartezeit beim nächsten Start entfällt. Bei sich ändernden Umständen ist ein Neustart des Systems notwendig, um z.B. neue Konfigurationen einzulesen.

7.1.2 Aufbau und Funktionsweise

Dieser Abschnitt befasst sich mit der Umsetzung der Spezifikation von Kapitel 5 in ein Projekt in ASCET. Es werden die einzelnen ASCET-Module vorgestellt.

Türschloss

Das Modul *DoorLock_Module* ist für die Zentralverriegelung zuständig.

[4]Bei einer Hysteresekurve erfolgt das Erkennen des höher liegenden Spannungspegels bei einem höheren Schwellenwert als beim Erkennen des niedrigeren Spannungspegels. Dadurch wird ein Hin- und Herflattern des Zustands bei kleinen Schwankungen vermieden.

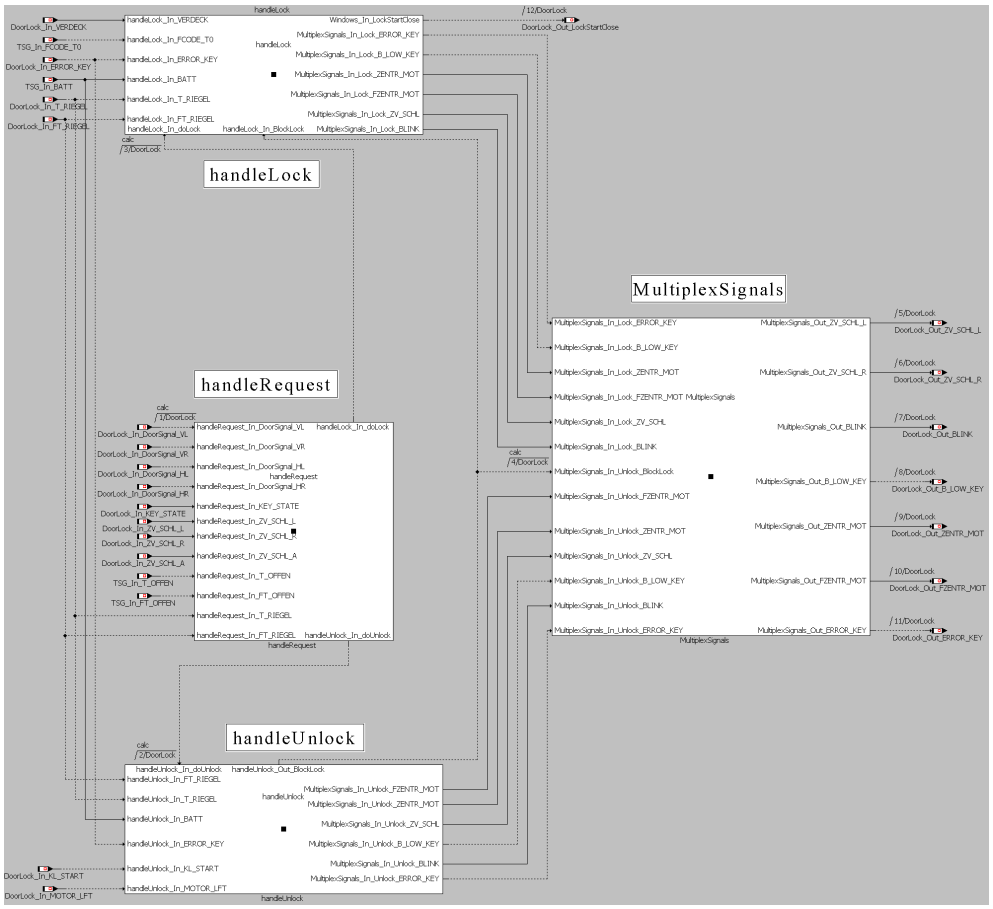


Abbildung 7.1: DoorLock_Module in ASCET.

handleRequest:

In dieser Komponente werden Befehle vom CAN-Bus, der aktuelle Türzustand und der Zustand des Türschlosses eingelesen und ausgewertet. Abhängig von den anliegenden Signalen wird, beim Eintreten von entsprechenden Bedingungen gemäss Abschnitt 5.2, der entsprechende Befehl weitergeleitet. *handleUnlock* erhält einen Befehl zum Öffnen, falls

- ein inkonsistenter Tür- und Riegelzustand vorliegt,
- eine Tür auf der Fahrzeugseite des TSGs geöffnet wird,
- das Türschloss aufgedreht wird oder
- eine CAN-Nachricht mit der Anweisung zum Öffnen eintrifft.

handleLock erhält einen Befehl zum Schliessen, falls alle Türen geschlossen sind und

- eine CAN-Nachricht mit der Anweisung zum Schliessen eintrifft oder
- das Türschloss zugedreht wird.

handleUnlock:

Das Eintreffen eines Öffnen-Befehls veranlasst *handleUnlock* den Entriegelungsvorgang (gemäss Abschnitt 5.2.2 ab dem Prüfen der Batteriespannung) durchzuführen. Dabei werden die Riegelmotoren nur gestartet, falls die Riegel nicht schon geöffnet sind.

handleLock:

Mit einem Schliess-Befehl von *handleLock* wird der Verriegelungsvorgang (ab der Überprüfung der Batteriespannung gemäss Abschnitt 5.2.2) angestossen. Wie bei *handleUnlock* wird nur dann eine Spannung an die Riegelmotoren gelegt, wenn die Endposition noch nicht erreicht worden ist.

Während des Bearbeitungsvorganges eines Befehls kann unter Umständen ein weiterer Befehl eintreffen, der den entgegengesetzten Verriegelungszustand wie der bisherige Befehl zum Ziel hat. *handleLock* und *handleUnlock* würden gleichzeitig die Motoren in verschiedene Richtungen drehen lassen. Dieser Konflikt wird aufgelöst, indem das Entriegeln Vorrang vor dem Verriegeln erhält. Als Vorkehrungsmassnahme zur Konfliktauflösung wird *handleLock* für die Zeit des Entriegelns blockiert und kann dabei keine Aktionen ausführen. Beim Eintreffen eines Öffnen-Befehls in *handleUnlock* wird *handleLock* darüber in Kenntnis gesetzt. *handleLock* wird durch einen hierarchischen Automaten mit zwei Oberzuständen implementiert. Die Logik für die normale Ausführung befindet sich in einem dieser beiden Zustände. Das Blockieren durch *handleUnlock* veranlasst den Wechsel in den anderen Zustand. Der Kontrollfluss kehrt erst nach Beendigung der Entriegelungsaktion in *handleUnlock* wieder in den ersten Oberzustand (und darin in den ursprünglichen Startzustand) zurück. Ein, in der Mitte abgebrochener Verriegelungsvorgang wird nicht mehr nachgeholt und muss neu angestossen werden.

Mit dieser Vorgehensweise werden inkonsistente Türzustände auf bei-

den Fahrzeugseiten auf Grund von Synchronisationsproblemen vermieden. Sobald die Türen auf einer Seite geöffnet werden, erhält das gegenüberliegende TSG über den CAN-Bus ebenfalls die Anweisung zum Öffnen.

MultiplexSignals:

Abhängig von der aktuell ausgeführten Aktion (Verriegeln oder Entriegeln) leitet *MultiplexSignals* nur Signale der betreffenden Komponente (*handleLock* oder *handleUnlock*) an den TSG-Ausgang weiter.

Fensterheber

Das Modul *Windows_Module* ist für den Fensterheber zuständig.

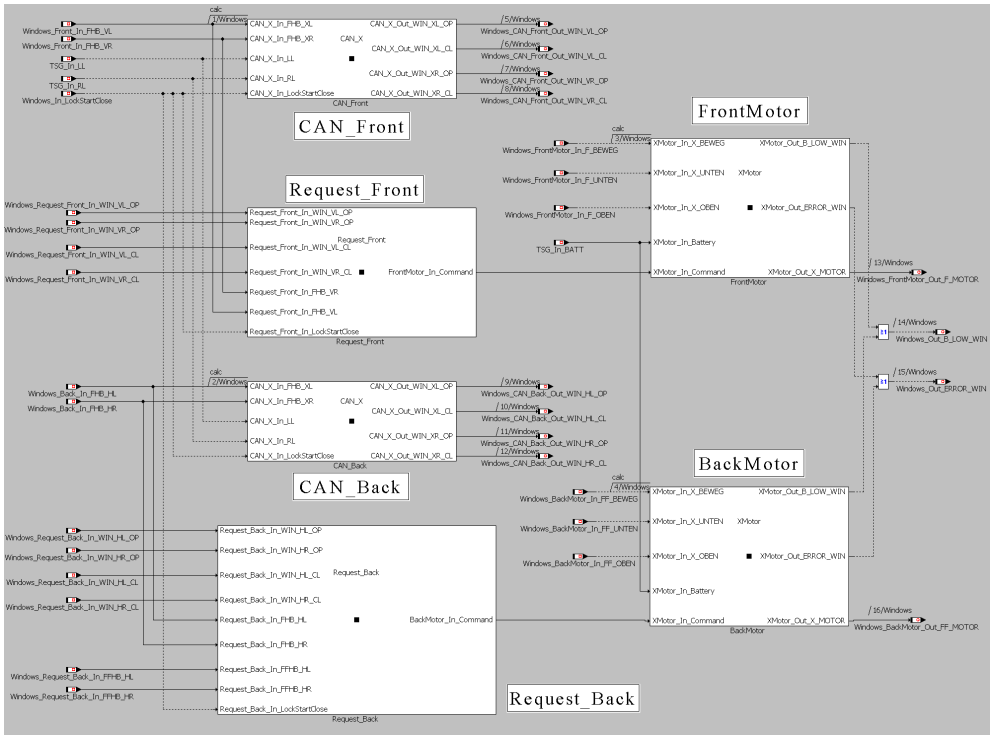


Abbildung 7.2: Windows_Module in ASCET.

CAN_Front bzw. CAN_Back:

Die Implementierungen von *CAN_Front* und *CAN_Back* sind identisch.

Beide registrieren eingehende Anweisungen des Fahrers zur Bewegung der Fenster auf der gegenüberliegenden Fahrzeugseite und leiten diese über den CAN-Bus weiter. Sie unterscheiden sich dahingehend, dass *CAN_Front* CAN-Nachrichten für das gegenüberliegende vordere Fenster generiert, während *CAN_Back* CAN-Nachrichten für das hintere Fenster auf der anderen Seite erzeugt. Damit einher geht auch eine unterschiedliche Interpretation der Eingangssignale, abhängig von der Konfiguration der Fahrer- und Fahrzeugseite.

In *CAN_Front* bzw. *CAN_Back* folgt auf das Einlesen der Eingangssignale ggf. eine Verzögerung um 0,5 Sekunden (vgl. Fussnote 2 auf Seite 50). Hinzu kommt die Auswertung des Signals von *Doorlock_Module*, dass beim Verriegeln der Türen an *Windows_Module* übermittelt wird. Eine entprechende TSG-Konfiguration führt damit zur Generierung eines Schliessbefehls. Am Ende der Signalbearbeitung werden die resultierenden Befehle für den CAN-Treiber aufbereitet.

Request_Front bzw. Request_Back:

Die Komponenten *Request_Front* und *Request_Back* verarbeiten Anweisungen zur Bewegung der Fensterscheiben auf derselben Fahrzeugseite wie das TSG. *Request_Front* ist für das vordere Fenster und *Request_Back* für das Hintere zuständig. Beide Komponenten arbeiten prinzipiell nach derselben Weise und unterscheiden sich hauptsächlich in der Art und Anzahl der Eingänge. *Request_Front* erhält die Anweisungen vom CAN-Bus, der vorderen Fensterhebertaste und von der Zentralverriegelungskomponente *Doorlock_Module*. *Request_Back* hingegen reagiert auf Anweisungen vom CAN-Bus, der hinteren Fensterhebertaste, der Fahrer-Fensterhebertaste und von *Doorlock_Module*. Bei den Fensterhebertasten wird dem Signal unter Umständen eine Verzögerung hinzugefügt (vgl.Fussnote 2 auf Seite 50).

Aus der Auflösung der vorbereiteten Eingangswerte folgt die resultierende Anweisung, die dann an die Motoransteuerung weitergeleitet wird. Dazu werden folgende Schritte zyklisch (bei jedem Aufruf durch den Scheduler) wiederholt:

1. überprüfe, ob eine manuelle Fensterbewegung[5] beendet werden soll

[5]vgl. Fussnote 2 auf Seite 50

und keine weitere Anweisung desselben Typs[6] mehr anliegt[7]

2. falls ja, dann leite einen Stopbefehl an die Motoransteuerung weiter

3.a) ansonsten überprüfe, ob es eine neue Anweisung (die zur Fensterscheibenbewegung führen soll) gibt; z.B. wenn ein bislang manuelles Fensterhochfahren in ein automatisches Fensterhochfahren übergehen soll, oder wenn z.B. statt eines manuellen Fensterhochfahrens ein manuelles Fensterrunterfahren erwünscht ist

3.b) falls ja, dann wähle diejenige mit der höchsten Priorität und leite den entsprechenden Befehl an die Motoransteuerung weiter

Auf diese Weise wird nach Ablauf der Schritte immer die zeitlich zuletzt getätigte Benutzeranweisung ausgeführt. Auch für den Konfliktfall, dass mehrere Anforderungen gleichzeitig eintreffen, ist durch die Priorisierung in Schritt 3.b) gesorgt.

Diese Schrittreihenfolge verhindert zudem, dass der Motor zu früh abgestellt wird:

Beispiel 7.1.7 *Der Fahrer auf der linken Fahrzeugseite drückt die Taste zum manuellen Runterfahren des Beifahrerfensters. Das rechte TSG erhält über den CAN-Bus die neue Anweisung zum Herunterfahren der vorderen rechten Fensterscheibe (Schritt 3.a)). Diese Anweisung wird an die Motoransteuerung weitergeleitet und führt zur Runterbewegung der Scheibe (Schritt 3.b)). Danach betätigt der Beifahrer auf der rechten Seite die Taste zum manuellen Runterfahren derselben Fensterscheibe (Anweisung vom selben Typ, wie die des Fahrers). Das TSG verarbeitet das Signal und leitet es an die Motoransteuerung weiter (Schritt 3.a) und 3.b)). Während sich die Scheibe noch in Bewegung befindet, lässt der Fahrer nun die Taste los und über den CAN-Bus gelangt das Signal „Neutrale Stellung“ an das Beifahrer-TSG. Da der Beifahrer aber noch die eigene Fensterhebertaste gedrückt hält, zeigt dies folgerichtig keine Wirkung (Bedingung in Schritt 1 trifft nicht zu). Erst wenn der Beifahrer seine Taste loslässt, wird die Bedingung in Schritt 1 erfüllt und Schritt 2 ausgeführt.*

[6] manueller Fensterbewegungsmodus mit gleicher Bewegungsrichtung

[7] Die Unterscheidung von Anweisungen erfolgt nach Eingang (CAN, Taste,...), Modus (manueller/automatischer Fensterbewegung) und Richtung (hoch bzw. runter)

FrontMotor bzw. BackMotor:

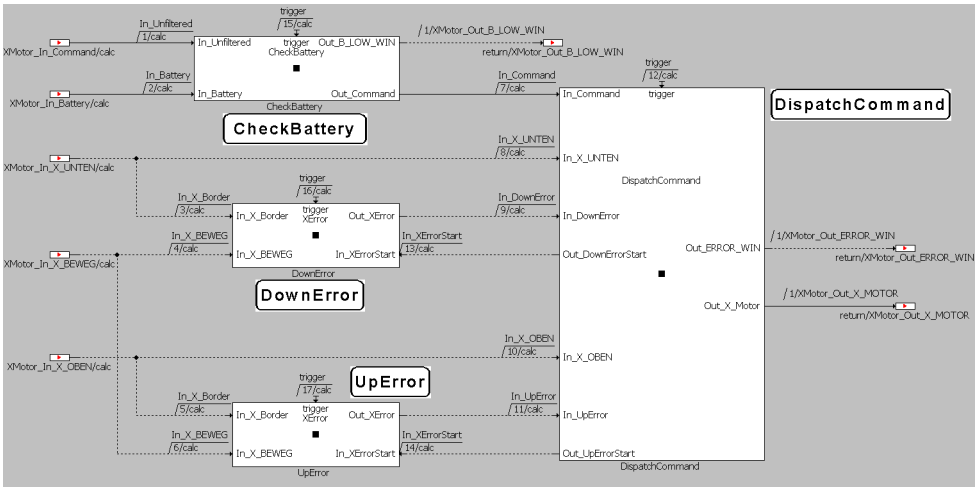


Abbildung 7.3: FrontMotor in ASCET.

Die Motoransteuerungen *FrontMotor* (siehe Bild 7.3) und *BackMotor* sind identisch implementiert. *FrontMotor* erhält die Anforderung von *Request_Front* und steuert den vorderen Fensterscheibenmotor an. Bei *BackMotor* und *Request_Back* gilt Analoges für die hintere Fensterscheibe. Bei ausreichender Batteriespannung wird die Motorbewegung durch das Anlegen der entsprechenden Spannung an die TSG-Ausgänge angestossen. Die Scheibenbewegung wird erst wieder gestoppt bzw. umgekehrt, wenn

- eine andere Anforderung von *Request_Front* bzw. *Request_Back* anliegt,
- die Scheibe den Anschlag erreicht hat,
- der 3 Sekunden Timer abgelaufen ist oder
- die Scheibe während der Bewegung blockiert.

Der letzte Fall aktiviert beim Scheibenhochfahren den Einklemmschutz. Die Scheibe wird dann solange wieder runtergefahren, bis sie am Anschlag angelangt ist, 3 Sekunden abgelaufen sind oder blockiert.

Bild 7.4 veranschaulicht das Verhalten der Komponente *DispatchCommand*. In den Zuständen *Open* und *Close* wird die Scheibe gemäss obiger Beschreibung Herunter- bzw. Hochgefahren. *Jam* repräsentiert den Zustand, in dem der Einklemmschutz aktiv ist.

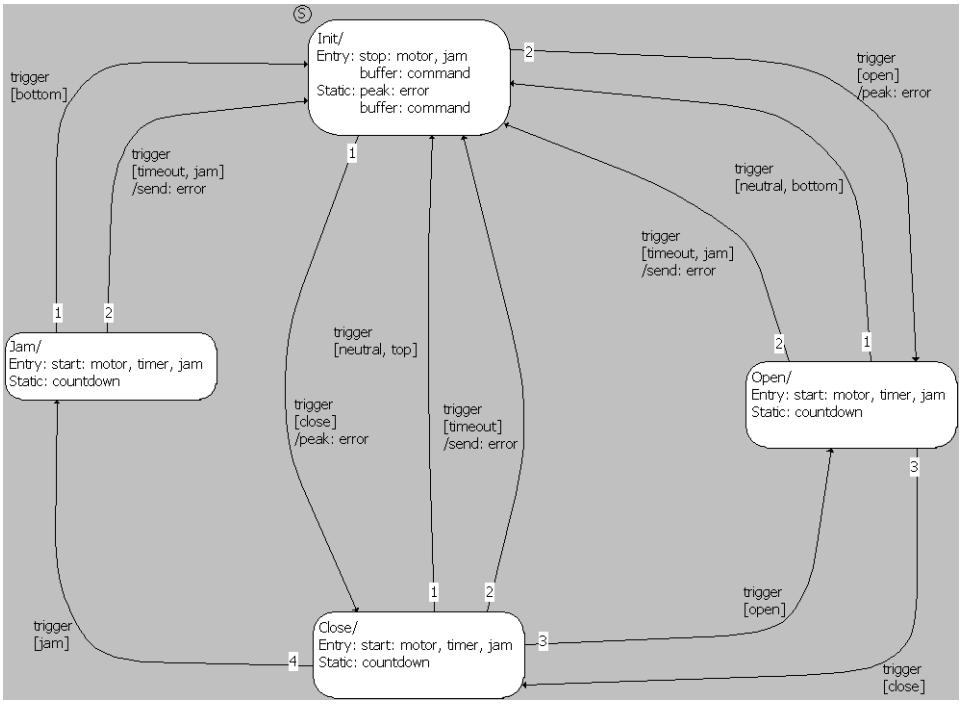


Abbildung 7.4: DispatchCommand in ASCET.

Beleuchtung

Die beiden Module *DoorLight* und *InteriorLight* sind für die Ausstiegsleuchten bzw. die Innenraumbeleuchtung zuständig. Sie setzen die, in Abschnitt 5.4 geforderte Funktionalität um.

Hilfsmodul

Gemäss der TSG-Spezifikation übermittelt das TSG auf der Fahrerseite dem gegenüberliegenden TSG den aktuellen Türstatus. Das Beifahrer-TSG kombiniert diesen Status mit dem Zustand der eigenen Türen und

sendet das Resultat zurück. Je TSG ist das Modul *DoorState* dafür zuständig, den kombinierten Türzustand zur Weiterverarbeitung an die Module *Doorlock_Module* und *InteriorLight* weiterzuleiten.

7.2 Testen auf Modellebene (Simulation)

Basierend auf dem erstellten ASCET-Modell der Türsteuerung erfolgte die Anwendung von konventionellen Qualitätssicherungsmassnahmen. Die Testverfahren können sowohl auf das Modell, als auch auf den daraus generierten C-Code angewandt werden. Für das Testen auf Modellebene stellt ASCET eine Simulationsumgebung zur Verfügung. Bei dem als „Offline-Experiment“ bezeichneten Verfahren erfolgt die Ausführung des generierten Codes auf dem PC. Durch eine entsprechende Einbettung des Codes kann der Programmablauf in der Simulationsumgebung visualisiert werden. Damit lassen sich Werte (Nachrichten, Variablen, aktueller Zustand in Automaten, etc.) mit Hilfe von numerischen und grafischen Anzeigen darstellen. Denkbar wäre auch eine Coverageanalyse der Automaten, wofür zum jetzigen Zeitpunkt allerdings keine Unterstützung in ASCET vorliegt. Eine Überdeckungsanalyse auf Modellebene wird zum Beispiel von Matlab/Simulink mit dem Simulink Performance Tool unterstützt [The].

Auf einen In-the-Loop-Ansatz wurde hier verzichtet, da für die überschaubaren Schnittstellen des TSGs zur Umgebung kein elaboriertes physikalisches Umgebungsmodell, wie z.B. für ein Fahrwerk, erforderlich war. Im vorliegenden Fall wurden hauptsächlich Tests auf den ASCET-Modulen durchgeführt. Ein Grund dafür ist die weitgehende Unabhängigkeit der Module untereinander.

Die Testfälle sind der Spezifikation nach folgendem Schema entnommen worden:

1. Extrahiere eine Anforderung aus der Spezifikation (z.B. Licht leuchtet nur 30 Sekunden, wenn alle Türen geschlossen worden sind und die Zündung aus ist)

2. Generiere einen Eingabesequenzpräfix, der zu dem Systemzustand führt, ab dem diese Anforderung erfüllt werden kann (z.B. Schliessen aller Türen und Ausschalten der Zündung)

3. Vervollständige die Sequenz mit Eingaben, mit denen die Anforderung erfüllt wird (z.B. Aufrechterhalten des Zustands der geschlossenen Türen und der ausgeschalteten Zündung für 30 Sekunden)

Dabei wurde berücksichtigt, dass der in Schritt 2 erreichte Systemzustand unter Umständen nicht eindeutig ist (z.B. Fahrzeugtyp ist entweder ein Zweitürer oder ein Viertürer). Solche Mehrdeutigkeiten wurden durch zusätzliche Testfälle abgedeckt. Ebenso wurden negative Testfälle untersucht, die absichtlich nicht den Anforderungen entsprechen.

7.3 Testen auf Codeebene

Auch die, in Abschnitt 3.2.2 besprochenen Verfahren lassen sich auf den von ASCET generierten Code anwenden. Insbesondere wurde hierbei das White-Box-Testen auf Basis von Überdeckungskriterien herangezogen. Um grosse Testdatenmengen zu bewältigen, ist der Einsatz von Werkzeugen, die die Überdeckungsanalyse durchführen, erforderlich. ASCET unterstützt von Haus aus keine Code-Coverageanalyse, so dass technische Eingriffe für die Anbindung von externen Coverage-Tools notwendig sind. Im Rahmen der Diplomarbeit wurde dazu IBM Rational Test RealTime verwendet. Weiterhin wurde auf CTE XL zurückgegriffen, um Testdaten grafisch zu erstellen. Bild 7.5 veranschaulicht die Werkzeugkette, die zum Einsatz kam.

Wie beim Testen auf der Modellebene, wurden hier ASCET-Module betrachtet. Mittels CTE XL werden die Eingabesequenzen für die Moduleingänge erstellt. Dazu wird in der CTE-Matrix jedem Eingang zu jedem betrachteten Zeitpunkt ein Wert zugewiesen. Die exportierte Eingabesequenz kann dann mit Hilfe eines Skriptes in ein für ASCET lesbares ASCII-Format gewandelt werden.
Bevor ein Test mit diesen Testdaten ausgeführt werden kann, muss zunächst der Code aus dem zu testenden ASCET-Modell generiert werden. Dieser kann dann für eigene Zwecke auch unabhängig von der ASCET-Toolkette extern weiterverarbeitet werden. Der, in ASCET-MD, ASCET-RP und ASCET-SE generierte Code unterscheidet sich jeweils hinsichtlich der Schnittstelle zur Hardware (siehe unten). Der Teil für die Funktionslogik ist identisch.

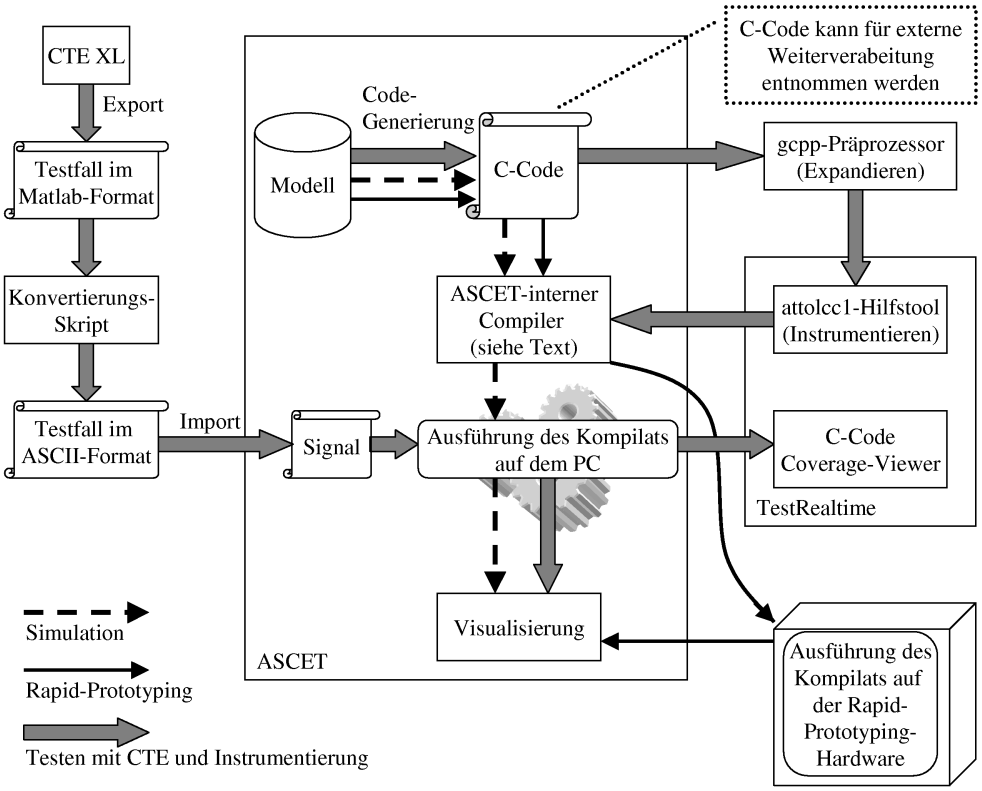


Abbildung 7.5: Testen mit ASCET.

Im Falle des Testens wird der Code zusätzlich mit Hilfstools expandiert[8], instrumentiert und kompiliert.
Abhängig vom ASCET-Werkzeug, werden unterschiedliche Compiler eingesetzt:

ASCET-MD: Für die Simulation und das Testen auf dem PC wird der Borland-C Compiler verwendet.

ASCET-RP: Für die Ausführung des Codes auf der Rapid-Prototyping-Hardware wird der Code für den PowerPC cross-kompiliert.

ASCET-SE: Für die Seriencodegenerierung stehen je nach Zielplattform (u.a. Infineon C16x und TriCore, Motorola MPC5xx und

[8] „attolcc1“ setzt C-Code voraus, in dem Makros expandiert und include-Dateien eingebunden sind.

M68HC12, TI TMS470, NEC V85x, Hitachi SH7055F) unterschiedliche Compiler zur Verfügung.

Im Rahmen des Testens wird die Simulationsumgebung (Messeinstellungen, Mappen der Testdaten auf die Moduleingänge) je Komponente einmal aufgesetzt und kann dann für weitere Testfälle wiederverwendet werden. Die Ausführung der Simulation geschieht auf Knopfdruck ohne weiteren Eingriff des Benutzers. Auf Grund der frei einstellbaren Zeitskalierung können die Simulationsberechnungen in kurzer Zeit durchgeführt werden. Die dabei entstehenden Ausgabedaten werden aufgezeichnet und können somit im Nachhinein analysiert werden bzw. stehen für Regressions- und Back-to-Back-Tests zur Verfügung.

Die Erstellung der Testdaten erfolgte zunächst nach demselben Schema wie in 7.2, um die grundlegende Funktionalität zu testen. Mit Hilfe der Codeüberdeckungsanalyse wurden dann Codestellen ausfindig gemacht, die noch nicht dem C_0-Kriterium bzw. der einfachen Bedingungsüberdeckung genügt haben. Da die Umsetzung von ASCET-Modellen (hier: fast ausschliesslich Zustandsautomaten) in C-Code ohne grössere Hilfskonstruktionen erfolgt, konnten die betreffenden Stellen den entsprechenden Modellteilen (Verzweigungsbedingungen, Transitionen, etc.) ohne Schwierigkeiten zugeordnet werden. Damit konnten dann weitere Testdaten erzeugt werden, um diese Stellen gezielt abzudecken.

7.4 Rapid-Prototyping

Viele Aspekte, wie z.B. das Fahrgefühl oder die Einwirkung einer realen Umgebung auf das Steuergerät können teilweise nicht oder nur unvollständig mit Hilfe einer Simulation erfasst werden. Eine Möglichkeit, dieses Problem während der Entwicklungsphase zu umgehen, stellt das Rapid-Prototyping dar. Wie schon im Abschnitt 3.2.3 angedeutet wurde, kann die Funktionalität damit frühzeitig getestet und abgesichert werden.

Für die Zwecke der Diplomarbeit standen eine Fahrzeugtür der Firma BMW und Rapid-Prototyping-Hardware der Firma ETAS zur Verfügung. Der Aufbau für das Rapid-Prototyping-Experiment wird in Bild 7.6 veranschaulicht.

Das eigentlich (in späteren Entwicklungsstufen) in der Fahrzeugtür ver-

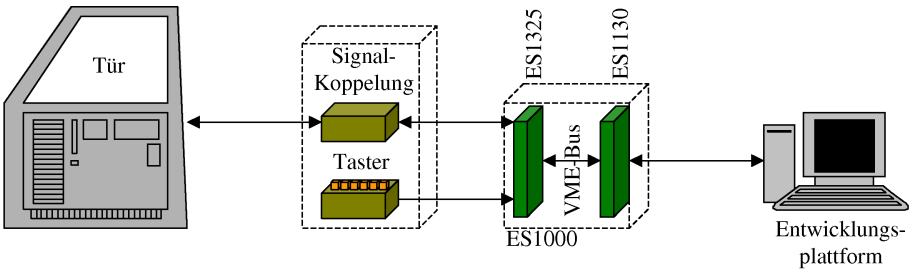


Abbildung 7.6: Rapid-Prototyping-Aufbau.

baute Türsteuergerät wird durch die Rapid-Prototyping-Hardware emuliert. Diese besteht aus dem ES1000 Gehäuse, welches mit verschiedenen Steckkarten den Umweltanforderungen angepasst werden kann. Im vorliegenden Fall sind darin die ES1130.3 und ES 1325 Karten verbaut, die beide über den VME-Bus miteinander verbunden sind. Die ES1130.3 Karte ist zuständig für die Kommunikation mit dem dem Entwicklungshost über eine Ethernet-Schnittstelle. Auf der Entwicklungsplattform wird mit ASCET-RP Code aus dem ASCET-Modell cross-kompiliert, auf die ES1130 geladen und dort ausgeführt. Die Interaktion mit Aktoren und Sensoren erfolgt dann über die ES1325 Karte. Dazu ist diese mit digitalen Ein- und Ausgängen ausgestattet. Allerdings werden damit die Aktoren bzw. Sensoren in der Tür nicht direkt angesteuert bzw. ausgelesen. Die Koppelung der Signalpegel erfolgt über eine separate Zwischenschaltung. Teilweise sind in diese Schaltung weitere Funktionen aufgenommen worden, die nicht durch die ETAS-Hardware nachgebildet oder aus technischen Gründen nicht in der Fahrzeugtür platziert werden konnten. So z.B. die Taster für den Fensterheber und die Aussenspiegeleinstellung. Desweiteren werden die Sensorsignale der Fensterscheibenbewegung in dieser Schaltung weiter aufbereitet:

Die Scheibenbewegung führt zur Drehung eines Rädchens, an dem zwei, um 90 Grad gedrehte Hallsensoren angebracht sind (siehe Bild 7.7).

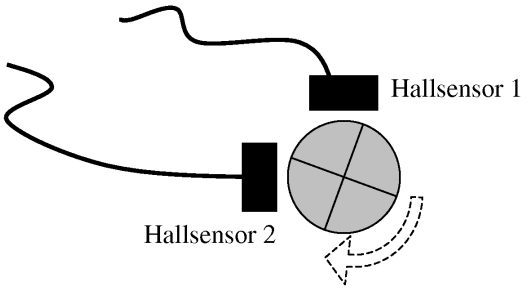


Abbildung 7.7: Hallsensoren zur Einklemmschutzerkennung.

Dadurch werden bei der Bewegung der Scheibe zwei, um 90 Grad verschobene Sinuskurven erzeugt. Die oben genannte Schaltung wandelt diese mit Hilfe von Schmitt-Triggern in zwei, um 90 Grad verschobene Rechteckkurven um (siehe Bild 7.8).

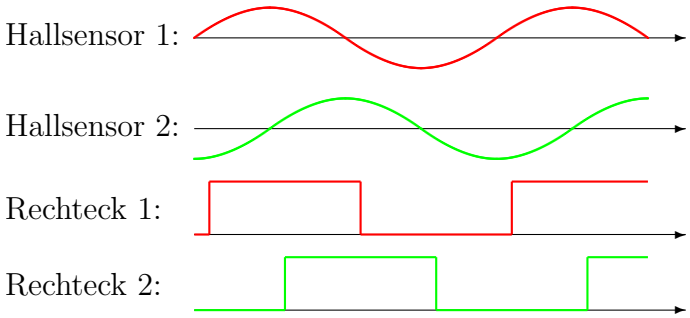


Abbildung 7.8: Signalverläufe der Sensoren.

Diese Rechteckkurven gelangen an die Eingänge der ES 1325 und können dann von der ES1130 weiter verarbeitet werden. Der zeitliche Abstand zwischen den Flanken der beiden Rechteckkurven deutet auf die Umdrehungsgeschwindigkeit des Rädchens und somit auf die Bewegungsgeschwindigkeit der Fensterscheibe hin. Überschreitet der Abstand zwischen zwei Flanken eine gewisse Schwelle, obwohl die Motoren eingeschaltet sind und sich die Fensterscheibe beim Hochfahren noch nicht am Anschlag befindet, so wird dies als Einklemmen interpretiert. Die Scheibe wird dann an den unteren Anschlag gefahren. Weiterhin kann man aus den Rechteckkurven die Fensterbewegungsrichtung ablesen und somit einen relativen Zähler implementieren (siehe Bild 7.9).

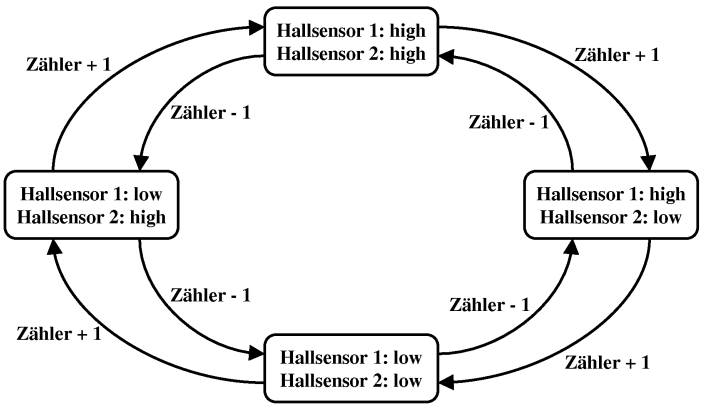


Abbildung 7.9: Zustandsbasiertes Quadrature Encoding.

Die Fraunhofer TSG-Spezifikation von Kapitel 5 konnte nicht als Grundlage dienen, da die dort gestellten Hardwareanforderungen hier nicht erfüllt wurden. Beispielsweise basiert der Einklemmschutz in der Fraunhofer Spezifikation auf Sensoren für den Anschlag der Fensterscheibe am Fensterrand. Hier hingegen sind solche Sensoren nicht vorhanden. Die Positionsbestimmung musste über den eben beschriebenen Zähler realisiert werden. Das hier implementierte Modell hält sich daher an die vorgegebene Hardware und erfüllt dessen Grundfunktionalitäten:

- Vertikales und horizontales Verstellen der Aussenspiegel mit Timeout
- Verriegeln und Entriegeln des Türschlosses
- manuelles bzw. automatisches Fensterhoch- und Fensterrunterfahren mit Einklemmschutz

Im Vergleich zur Entwicklung von Prototypen auf Basis der Zielplattform kann die Validierung beim Rapid-Prototyping ohne Rücksichtnahme auf Ressourcen-Beschränkungen erfolgen. Auch der Eingriff in das System durch Messungen ist möglich, insofern Echtzeiteigenschaften nicht davon betroffen sind. In ASCET-RP können beim Rapid-Prototyping in einem „Online-Experiment“ Zustandsänderungen des Systems rückwirkend im Modell auf der Entwicklungsplattform dargestellt werden.

Kapitel 8

Modellierung in AutoFocus

Dieses Kapitel beschreibt zunächst die Umsetzung der Fraunhofer Spezifikation des Türsteuergerätes in AutoFocus. Im Anschluss daran wird die Verifikation mittels Modelchecking als Anwendung eines formalen Verifikationsverfahrens erläutert.

8.1 Modell des Türsteuergeräts

8.1.1 Rahmenbedingungen

Aus denselben Gründen wie in Abschnitt 7.1.1 werden zunächst die Rahmenbedingungen definiert, bevor darauf basierend auf das TSG-Modell eingegangen wird.

Schnittstellenkonvention

In AutoFocus kommuniziert das Türsteuergerät über Kanäle mit der Umgebung (vgl. Abschnitt 6.3.1). Den Kanälen sind Datentypen zugeordnet, die die Signalwerte repräsentieren. Datentypen können Standarddatentypen (z.B. Bool oder Int) oder benutzerdefinierte Datentypen (z.B. WIN_CAN) sein. Die kanonische Modellierung der TSG-Schnittstelle zur Umgebung basiert auf folgender Konvention:

	Eingänge	Ausgänge
Signale über S1, S2[1]	Sensoren liefern bei jedem Systemtick ein Signal; Kanalwert entspricht immer dem aktuell gelesenen Sensorwert	Aktuatoren erhalten bei jedem Systemtick ein Stellsignal vom TSG; Kanalwert entspricht immer dem aktuellen Stellwert
periodische Signale über CAN-Treiber[2]	wenn bei einem Systemtick eine CAN-Nachricht anliegt, erhält entsprechender Kanal diesen Wert	CAN-Treiber versendet CAN-Nachrichten in periodischem Zeitraster (gemäss Fraunhofer Spezifikation); dazu wird letzter Wert ($\neq$ NoVal) des entsprechenden Kanals verwendet
spontane Signale über CAN-Treiber[3]	wenn bei einem Systemtick eine CAN-Nachricht anliegt, erhält entsprechender Kanal diesen Wert	für einen Systemtick liegt entsprechender Wert am Kanal an

Tabelle 8.1: Schnittstellenkonvention in AutoFocus.

Annahmen an die Umgebung

Die oben definierte Schnittstelle muss von der Umgebung eingehalten werden. Notfalls ist diese Voraussetzung durch den Einsatz von Wrapper-Komponenten zur erfüllen. Diese bilden dann das geforderte Schnittstellenverhalten der Umgebung auf die obige Schnittstellendefinition ab. Eine konkrete Schnittstelle ist stark implementationsabhängig und korreliert z.B. mit den Echtzeiteigenschaften und dem Task-Scheduling (z.B. von OSEK). Daher erfolgt hier die Konzentration auf die Hauptlogik gemäss der TSG-Spezifikation und eine Abstraktion wie beim ASCET-Modell.

Analog zu Abschnitt 7.1.1 müssen die Konfigurationswerte (z.B. RL oder FCODE_T0) vor Inbetriebnahme der Funktionen zunächst über den CAN-Bus eingelesen werden.

[1] Ein leerer Kanal (= NoVal) wird hier als gültiger Wert interpretiert, um der Zustandsraumexplosion beim Modelchecking entgegenzuwirken.

[2] Ein leerer Kanal (= NoVal) bedeutet hier, dass keine CAN-Nachricht anliegt.

[3] siehe Fussnote 2

8.1.2 Aufbau und Funktionsweise

Der Aufbau und die Funktionsweise gleichen prinzipiell dem ASCET-Modell aus Abschnitt 7.1.2, so dass hier im Wesentlichen die Unterschiede zu dessen Beschreibung hervorgehoben werden.

Die Funktionen werden auf die Hauptkomponenten *Windows*, *DoorLock* und *Light* verteilt.

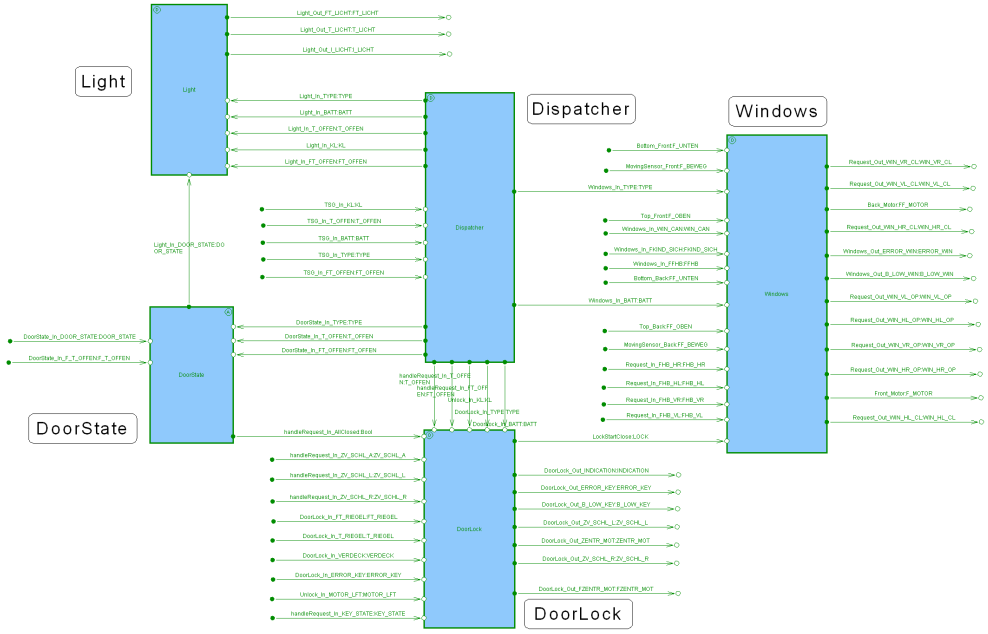


Abbildung 8.1: Überblick über das AutoFocus-Modell.

Türschloss

Die Komponente *DoorLock* ist für die Zentralverriegelung zuständig (siehe Bild 8.2). *SignalDispatcher* verteilt die eingehenden Signale an die anderen Komponenten von *DoorLock*.

handleRequest:

Befehle vom CAN-Bus, der aktuelle Türzustand und der Zustand des Türschlosses werden hier eingelesen und ausgewertet. Die Unterkompo-

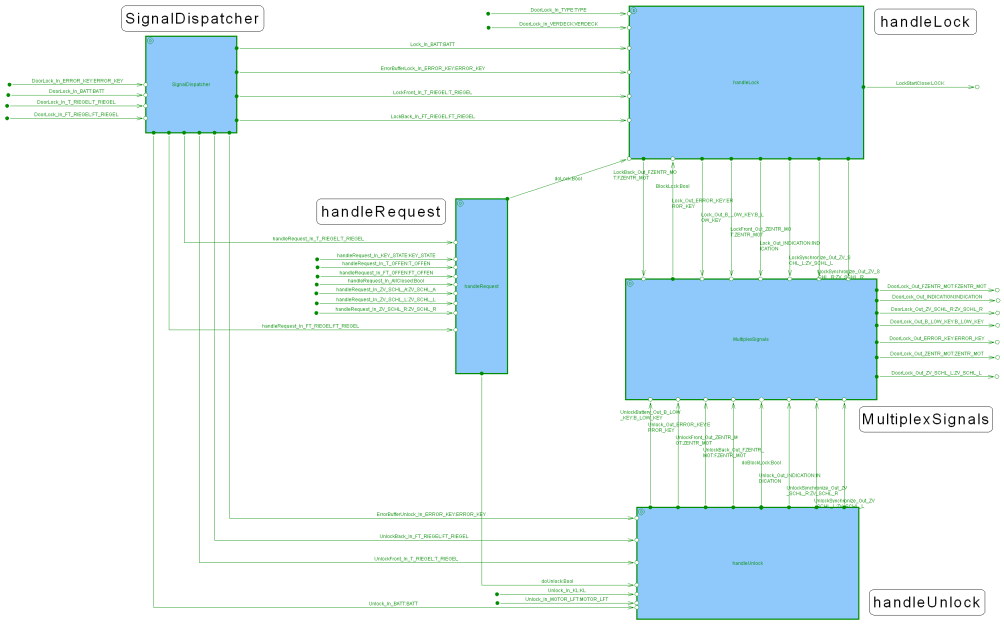


Abbildung 8.2: DoorLock-Komponente in AutoFocus.

nenten *RequestLock* und *RequestUnlock* lösen die Anforderungen auf und leiten den Befehl dann an *handleLock* bzw. *handleUnlock* weiter.

handleUnlock bzw. handleLock:

Die Beschreibung von Abschnitt 7.1.2 trifft hier ebenfalls zu. Im Unterschied zum ASCET-Modell werden jedoch parallel ausgeführte Aktivitäten in eigene Subkomponenten ausgelagert:

- Fehlernachrichten, die vom gegenüberliegenden TSG über den CAN-Bus verschickt werden, werden in *ErrorBufferUnlock* bzw. *ErrorBufferLock* gepuffert.
- Die Countdown-Uhr für den Timeout wird in *UnlockTimer* bzw. *LockTimer* ausgelagert.
- Die Ansteuerung der Motoren befindet sich in *UnlockBack/UnlockFront* bzw. *LockBack/LockFront*.
- Die Synchronisation mit dem Nachbartürsteuergerät übernimmt *UnlockSynchronize* bzw. *LockSynchronize*.
- *CloseWindows* stösst bei entsprechender TSG-Konfiguration das

Schliessen der Fenster an.

Konflikte bei gleichzeitigen Öffnen- und Schliessbefehlen werden hier ebenso umgangen, indem die Komponente *handleLock* beim Eintreffen eines Öffnenbefehls blockiert wird. Zum Zeitpunkt der Erstellung der Diplomarbeit wurden noch keine hierarchischen STD-Zustände von der Modelchecking-Anbindung unterstützt. Daher wurden die beiden Modi „Blockiert“ und „Unblockiert“ in *Lock* auf einen Automaten ohne Unterzustände abgebildet.

MultiplexSignals:

Die, von *handleLock* und *handleUnlock* erzeugten Signale werden über *MultiplexSignals* ausgegeben. Während einer Öffnenaktion werden nur die Signale von *handleUnlock* weitergeleitet. Ansonsten diejenigen von *handleLock*.

Fensterheber

Die Komponente *Windows* ist für den Fensterheber zuständig (siehe Bild 8.3). *Dispatcher* verteilt die eingehenden Signale an die betreffenden Komponenten.

CAN_Front bzw. CAN_Back:

Beide Komponenten funktionieren auf dieselbe Weise wie der entsprechende Teil des ASCET-Modells. Daher wird an dieser Stelle auf Abschnitt 7.1.2 verwiesen.

Request_Front bzw. Request_Back:

Request_Front und *Request_Back* unterscheiden sich in der Art und Anzahl der Eingänge. Beide Komponenten verarbeiten Benutzeranweisungen nach derselben Arbeitsweise wie in Abschnitt 7.1.2. Im ASCET-Modell erfolgt die Auflösung der Anweisungen mittels ESDL-Code (siehe Abschnitt 6.1.1). Dieses Beschreibungsmittel steht in AutoFocus nicht zur Verfügung, so dass das gleiche Verhalten mit Hilfe eines Komponentennetzwerkes nachgebildet wird. Die Verarbeitung erfolgt in Form einer Pipeline. Neue Signale durchlaufen dabei folgende Schritte:

1. Eingehende Signale werden abhängig von der TSG-Konfiguration

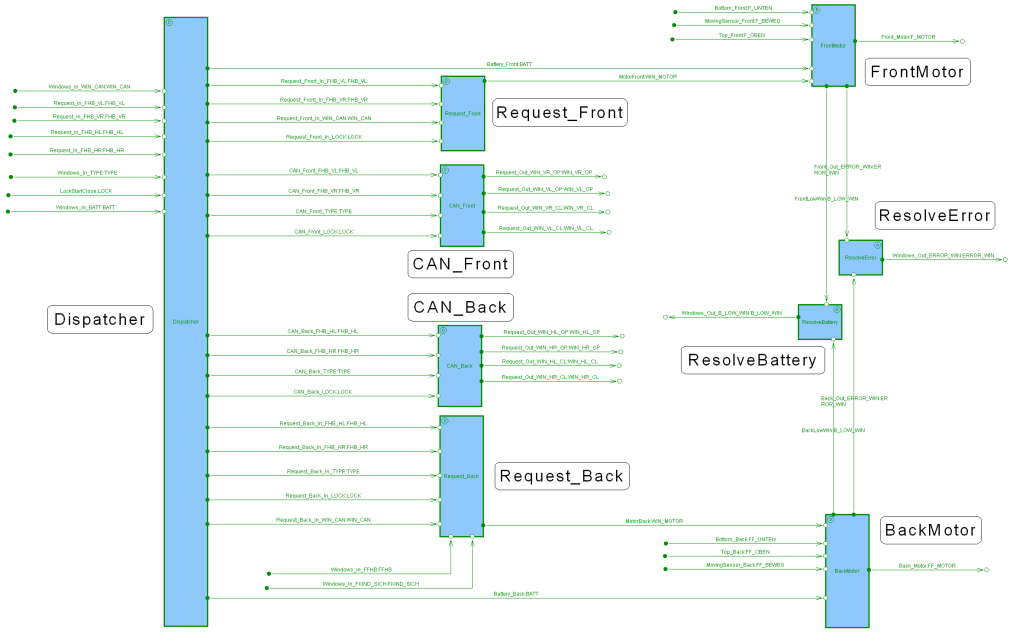


Abbildung 8.3: Windows-Komponente in AutoFocus.

ausgefiltert und aufbereitet[4].

2. Von neu eingetroffenen Anweisungen wird in der Unterkomponente *DispatchRequest* diejenige mit der höchsten Priorität ausgewählt.

3. Underscheidung der resultierenden Anweisung aus Schritt 2:

   a) Anweisung zur Fensterbewegung: Es wird das Flag für den Typ[5] dieser Anweisung gesetzt. Bisherig gesetzte Flags von anderen Anweisungstypen bleiben nur dann weiterhin gesetzt, wenn sie die gleiche manuelle Fensterbewegung darstellen (d.h. manuelles Fensterhochfahren bzw. manuelles Fensterrunterfahren) und die dazugehörigen Anweisungen immer noch am TSG-Eingang anliegen.

   b) Anweisung zur Neutralstellung: Das Flag für den Typ der Anweisung wird nur zurückgesetzt, falls der Typ eine manuelle Fensterbewegung darstellt.

[4]vgl. Fussnote 2 auf Seite 50
[5]vgl. Fussnote 7 auf Seite 74

4. Die gesetzten Flags führen zur gleichen Bewegungsrichtung des Fensters. Der entsprechende Befehl wird an die Motoransteuerungen *FrontMotor* bzw. *BackMotor* weitergeleitet.

Die Priorisierung in Schritt 2 löst Konflikte bei gleichzeitigem Eintreffen von neuen Anweisungen auf. Zudem führt dieses Vorgehen zum gewünschten Verhalten beim Loslassen von mehreren gedrückten Tasten in bestimmten Fällen:

Beispiel 8.1.1 *In diesem Kontext wird Beispiel 7.1.7 zum Vergleich wiederholt aufgegriffen: Der Fahrer auf der linken Fahrzeugseite drückt die Taste zum manuellen Runterfahren des Beifahrerfensters. Das rechte TSG erhält über den CAN-Bus die neue Anweisung zum Runterfahren der vorderen rechten Fensterscheibe (Schritt 1). Da nur diese einzige Anweisung zu diesem Zeitpunkt anliegt, ist dies die resultierende Anweisung von Schritt 2, dessen Flag in Schritt 3.a) dann auch gesetzt wird. Sie ist vom Typ „manuelle Fensterrunterbewegung vom CAN-Eingang“. Der entsprechende Befehl wird im Schritt 4 an die Motoransteuerung weitergeleitet und führt zur Runterbewegung der Scheibe. Danach betätigt der Beifahrer auf der rechten Seite die Taste zum manuellen Runterfahren derselben Fensterscheibe. Das TSG verarbeitet das Signal (Schritt 1) und registriert dieses als einzige neu eintreffende Anweisung (Schritt 2). Im 3.a) Schritt wird dessen Typ daher gemerkt (Typ: „manuelle Fensterrunterbewegung vom Taster“). Die bisherige Anweisung von der Fahrerseite hat denselben Typ und liegt immer noch an. Daher bleibt dessen Flag weitherhin gesetzt. Im Schritt 4 werden beide Anweisungen aufgelöst und erzeugen den Befehl für die Motoransteuerung zum Runterfahren. Während sich die Scheibe noch in Bewegung befindet, lässt der Fahrer nun die Taste los. Über den CAN-Bus gelangt das Signal „Neutralstellung“ an das Beifahrer-TSG und löscht das entsprechende Flag (Schritt 3.b)). Da der Beifahrer aber noch die eigene Fensterhebertaste gedrückt hält, bleibt das Flag der anderen bisher vermerkten Anforderung (vom Beifahrer) weiterhin vermerkt. Das Fenster wird weiter runtergefahren (Schritt 4). Erst wenn der Beifahrer seine Taste loslässt, sind nach Schritt 3.b) keine Flags mehr gesetzt und der Motoransteuerungsbefehl wird auf „Neutralstellung“ gesetzt.*

FrontMotor bzw. BackMotor:

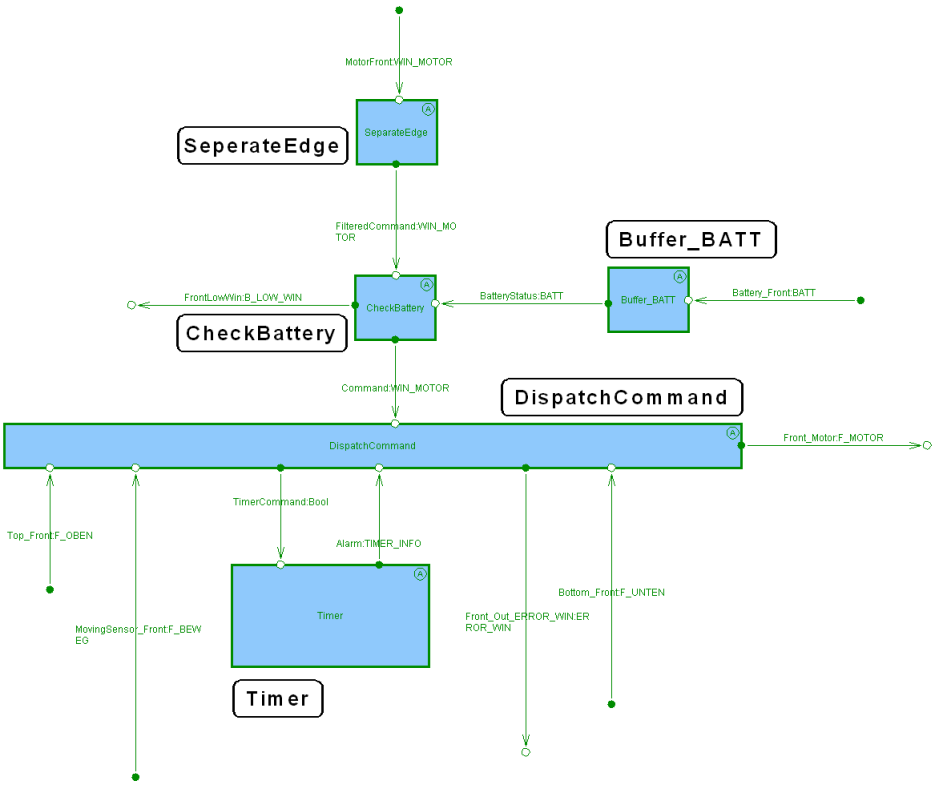


Abbildung 8.4: FrontMotor in AutoFocus.

Beide Komponenten sind äquivalent aufgebaut. Beim Eintreffen eines Befehls von *Request_Front* (siehe Bild 8.4) bzw. *Request_Back* wird der Motor durch die Unterkomponente *DispatchCommand* (siehe Bild 8.5) angesteuert. Nach dem Einschalten des Motors benötigt das Fenster auf Grund dessen Trägheit ein wenig Zeit, bis es in Bewegung gesetzt wird. Während dieser Zeit soll das Stehenbleiben der Scheibe jedoch nicht als Einklemmen bzw. Blockieren interpretiert werden. Dazu wird das Flag „StartupMode“ abgefragt. Zu Beginn einer Scheibenbewegung wird das Flag gesetzt. Nach Ablauf der Zeitspanne für das Anlaufen wird StartupMode wieder zurückgesetzt.

Die Unterkomponente *Timer* führt zum Abbruch der Scheibenbewegung bei einem Timeout. Auf Grund der Semantik von AutoFocus weist die Übermittlung von Nachrichten über Kanäle einen Pipeline-Effekt auf:

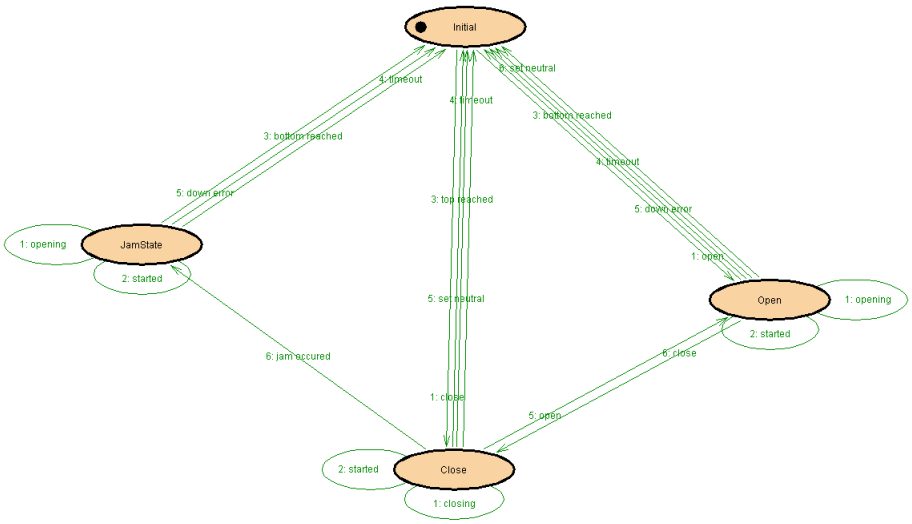


Abbildung 8.5: DispatchCommand in AutoFocus.

Ein Signalfluss durch mehrere Komponenten steht erst um mehrere Systemticks verzögert am Ende an. Verzweigt sich der Signalfluss, um nach einer parallelen Bearbeitung am Ende wieder zusammengeführt zu werden, so liegen beide Resultate unter Umständen nicht mehr zum gleichen Zeitpunkt bei der Zusammenführung an.
Somit kann z.B. der Fall eintreten, dass das bisherige Öffnen des Fensters zu einem Timeout führt und gleichzeitig ein neuer Befehl zum Schliessen des Fensters in *DispatchCommand* eintrifft. Der neu begonnene Schliessvorgang wird dann fälschlicherweise durch den verzögerten Timeout-Alarm abgebrochen. Mit Hilfe des Flags „IgnoreTimeout“ in *DispatchCommand* wird dies unterbunden. Während des Zeitintervals, in dem ein verzögerter Alarm eintreffen kann, ist dieses Flag gesetzt. Bei gesetztem Flag wird ein Timeout-Alarm nicht bearbeitet und ist daher wirkungslos.

Beleuchtung und weitere Teilaufgaben:

Zu den Komponenten *DoorLight*, *InteriorLight* und *DoorState* wird auf die Beschreibung in Abschnitt 5.4 verwiesen.

8.2 Testverfahren

Im Rahmen der konventionellen Qualitätssicherung bietet sich in AutoFocus die Simulationsumgebung SimCenter an. Der Ablauf der Berechnungen der einzelnen SSDs und STDs eines Systems lässt sich hierbei visualisieren. Über einen Editor können je Systemtick die Eingabekanäle des Modells mit Daten belegt werden. Testdaten können somit manuell eingegeben werden, um Funktionen gezielt überprüfen zu können. Andererseits stellt das AutoFocus-Framework mit dem „Test Input Generator“ (TIG) eine Möglichkeit zur Verfügung, Testdaten automatisch generieren zu lassen, die dann z.B. für die Simulation als Eingabedaten zur Verfügung stehen. Der TIG analysiert die Schnittstelle des Modells zur Umgebung und gruppiert die Eingangskanäle. Zu jeder Gruppe werden alle möglichen Tupel von Eingabewertekombinationen generiert und nacheinander im „Test Data Format“ abgespeichert. Die TDF-Datei kann dann von SimCenter eingelesen werden. Folgende Varianten zur Erzeugung von Testdaten bietet der TIG hierbei an:

Simple Input Generator: Jeder einzelne Eingabekanal wird als Gruppe betrachtet.

Pair Input Generator: Jede Paarkombination von zwei Eingabekanälen wird hier einmal als Gruppe betrachtet.

Connected Input Generator: Die Eingangskanäle von Komponenten, die mit dem <<connected>>-Attribut gekennzeichnet sind, werden jeweils zu einer Gruppe zusammengefasst.

All Input Generator: Hiermit werden alle möglichen Eingabewertekombinationen getestet.

Configurable Input Generator: Mit Hilfe des „Group Configuration Formats“ (GCF) kann man die Testdaten individuell spezifizieren. Der TIG erzeugt daraus die TDF-Datei.

Mit Hilfe des Codegenerators der Firma Validas [Val] lässt sich aus dem zu testenden AutoFocus-Modell C-Code erzeugen. Das daraus erstellte Programm kann mit den TIG-Testdaten ausgeführt werden. Ausgabe und Eingabe zusammen bilden dann Testfälle.

Bei der modellbasierten Entwicklung können diese Testfälle dann auf Verfeinerungen bzw. Implementierungen des AutoFocus-Modells ange-

wandt werden. Dazu müssen die Testfälle vorher an das neue Modell angepasst bzw. konkretisiert werden. Bild 8.6 stellt diesen Vorgang des modellbasierten Testens dar.

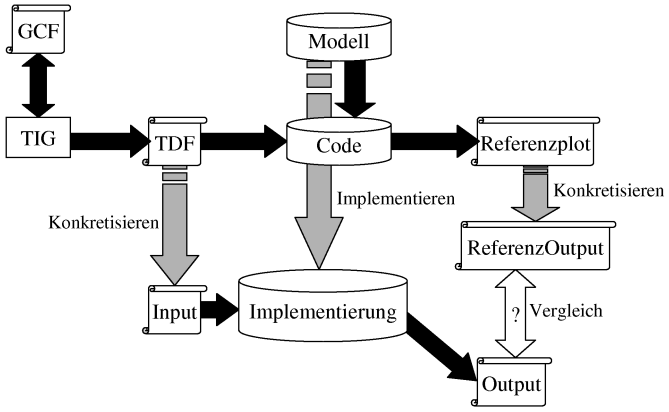


Abbildung 8.6: Modellbasiertes Testen mit AutoFocus.

Bei Back-to-Back-Tests werden die Ausgaben miteinander verglichen, um bei Abweichungen Fehler feststellen zu können. Voraussetzung für einen effizienten Ablauf ist ein automatisierter Ansatz.

Liegt als Testobjekt ein transformationelles System (vgl. Abschnitt 4.2) vor, so stellt der TIG ein wirkungsvolles Instrument dar, um ohne grossen Aufwand Testdaten aus dem Testobjekt zu generieren. Die erzeugten Testdaten liegen in sequentieller Form vor und können nacheinander in das Testobjekt eingegeben werden. Jeder Schritt entspricht einem Testlauf.

Schwieriger gestaltet sich die Lage bei reaktiven Systemen. Hier hängt die Ausgabe von der vorher getätigten Eingabe und vom bisherigen Zustand ab. Ein Testlauf ist hier nicht eine einmalige Eingabe, sondern eine (aus praktischen Gründen beschränkte) Sequenz von Eingabedaten. Die Schwierigkeit liegt darin, eine inhaltlich zusammenhängende Testsequenz zu erzeugen, die möglichst viele Systemzustände abdeckt (vgl. Abschnitt 3.2.2). Eine willkürliche Testdateneingabe führt in endlicher Zeit mit geringer Wahrscheinlichkeit zu aussagekräftigen Testdatensequenzen. Was durch den Einsatz des TIG somit nicht automatisiert wird, ist das Erstellen von GCF-Testspezifikationen, die dem Ablauf eines reaktiven Systems eine Richtung vorgeben.

Eine andere Möglichkeit, Testsequenzen für reaktive Systeme zu erstellen bietet sich mit dem Modelchecker an. Um Beispielsweise eine Eingabesequenz zu erhalten, die das System in einen definierten Zustand bringt, kann eine entsprechend negierte Formel dem Modelchecker übergeben werden: „Zielzustand kann nie erreicht werden.“ Da der erwünschte Zustand jedoch erreicht werden kann (insofern das Modell diesbezüglich korrekt ist), liefert der Modelchecker ein Gegenbeispiel. Dieses kann als Testsequenz für die Implementierung verwendet werden.

Im Gegensatz zum Testen, basierend auf Modellen, ermöglicht der Validas Codegenerator auch das Testen auf Codeebene. Es können die in 3.2.2 erwähnten Verfahren auf den generierten Code angewandt werden. Die Umsetzung der AutoFocus-Modelle in Code erfolgt beim Validas Codegenerator ähnlich wie beim ASCET-Codegenerator auf eine kanonische Weise. Daher kann man von Fehlern im Code leicht auf die entsprechenden Modellteile schliessen, um dort die Fehler zu beheben.

8.3 Modelchecking

Komponenten ab der Grössenordnung von *Windows* oder *DoorLock* wurden im Rahmen der Diplomarbeit nicht mehr verifiziert. Grund war nicht der dafür notwendige Speicheraufwand, sondern der Zeitaufwand von SMV. Beispielsweise war SMV mit dem Nachweis der, in Abschnitt 8.3.1 vorgestellten Eigenschaft für *Windows* nach 3 Stunden immer noch nicht fertig mit der 4. Fixpunkt-Iteration. Im Vergleich dazu benötigte der Nachweis der Eigenschaft für die Subkomponente *FrontMotor* mit über 400 Iterationen unter 4 Minuten.

SMV wurde daher vorrangig für die Verifikation von Eigenschaften von Teilkomponenten mit weniger als 10 STDs verwendet. Damit konnten logische Fehler ausgeschlossen bzw. leicht ausgemacht werden.

Beispiel 8.3.1 *Um zu überprüfen, ob eine Implikation „$A \Rightarrow B$“ auch tatsächlich zur Anwendung kommen kann, kann die Erreichbarkeit von Zuständen abgefragt werden, in denen die Prämisse „A“ erfüllt ist: „AG not(A)“ liefert ein Gegenbeispiel, das die Erreichbarkeit eines Zustands, in dem „A“ erfüllt wird, belegt.*

Allerdings schwankte der zeitliche Aufwand abhängig von der Komple-

xität der Komponenten zwischen 2 bis 10 Minuten. Um den zeitlichen Aufwand beim Modelchecking zu verringern, wurde der Zustandsraumexplosion durch folgende Massnahmen bei der Modellierung entgegengewirkt:

- Schnittstellen des TSGs zur Umgebung wurden so modelliert, dass ein leerer Kanal (NoVal) ebenso als Wert interpretiert wird[6]:
  Statt z.B. einem Kanal den Datentyp „Bool“ mit der Wertemenge {False, True} zuzuweisen, wird ein benutzerdefinierter Datentyp „Foo“ mit der Wertemenge {Bar} verwendet. Dadurch reduziert sich die Wertemenge {NoVal, False, True}, mit der ein Kanal belegt werden kann auf die Wertemenge {Noval, Bar}.

- Trotz der Definition eines Limits bei der Verwendung von Integerwerten (MaxInt), können mehrere, unabhängig verwendete Integervariablen den Zustandsraum erheblich vergrössern. Beispielsweise werden in dem TSG-Modell Timerkomponenten eingesetzt, die nach Ablauf eines Countdownzählers (in Form einer Integervariablen) einen Alarm ausgeben. Werden z.B. 2 solcher Timerkomponenten eingesetzt und ist MaxInt auf 15 gesetzt, so vergrössert sich der Zustandsraum u.U. um den Faktor 15^2. Besteht zwischen beiden Timerkomponenten eine Ausschlussbedingung (z.B. wenn auf Grund des Modellverhaltens immer nur ein Countdown-Timer aktiv ist), so können beide Komponenten zu Einer zusammengefasst werden. Somit reduziert sich der Faktor auf den Wert 15. Eine weitere Verbesserung kann erzielt werden, indem der Countdownzähler auf einen definierten Wert gesetzt wird (z.B. -1), falls der Zähler gerade nicht verwendet wird. Dies verringert ebenfalls die Anzahl der möglichen Werte, die der Zähler in Kombination mit den restlichen Systemzuständen annehmen kann.

Im Sinne von „Design for Testability“ könnte man hier den Begriff „Design for Modelcheckability“ ansetzen, wo schon bei der Modellierung eines Systems hinsichtlich des Zustandsraumproblems gewissen Empfehlungen Folge geleistet werden könnte.

Das Modell des TSGs greift nicht auf Immediate Ports zurück. Somit erfolgen Berechnungsschritte der Komponenten auch nicht in einem unendlich kleinen Zeitintervall. Dies genügt dem Umstand, dass ein Berech-

[6] vgl. Abschnitt 8.1.1

nungsschritt auf einem Mikroprozessor ebenfalls Zeit benötigt. Als Konsequenz ergibt sich ein pipeline-artiges Verhalten der Berechnungsschritte, was analog zum integrierten Schaltungsentwurf zu Hazards führen kann (vgl. Abschnitt 8.1.2). Damit sich die einzelnen Signale nicht überholen, können Synchronisationsmassnahmen wie z.B. Zwischenpuffer dem entgegen wirken.

Im Unterschied zu einem rein event-gesteuerten System, bei dem eine Berechung nur in Folge des Eintreffens eines äusseren Ereignisses stattfindet, existieren im TSG-Modell auch zeitliche Abhängigkeiten. Beispiele sind die zahlreichen Timeouts und Countdown-Timer. Da AutoFocus einen diskreten Systemtick zu Grunde legt, können keine kontinuierlichen Zeitwerte betrachtet werden. Mittels Integervariablen werden Countdownzähler modelliert. Die Genauigkeit der Zeitauflösung wird durch die Anzahl der Zählschritte bestimmt. Zum Beispiel könnte 1 Sekunde durch 10 Systemticks dargestellt werden. Es muss ein Kompromiss zwischen der Genauigkeit der Auflösung und dem Zeitaufwand für das Modelchecking getroffen werden.

8.3.1 Verifikation des Einklemmschutzes

Ein durchgängiges Beispiel für den Einsatz von Modelchecking ist der Nachweis der Korrektheit des Einklemmschutzes. Wie schon oben erwähnt, konnte der Modelchecker das Gesamtmodell nicht auf einmal verarbeiten. Daher wurde ein modularer Ansatz gewählt, um die Korrektheit des Einklemmschutzes für das Gesamtsystem zu zeigen. Die, für den Einklemmschutz bedeutsame Komponente *FrontMotor* befindet sich innerhalb der *Windows*-Komponente. Hier wird die Eigenschaft exemplarisch für das Vorderfenster gezeigt. Ein analoges Vorgehen ergibt sich für das Hinterfenster.

Die Schnittstelle dieser Komponente besteht aus den Eingängen:

- *Bottom_Front*: registriert, ob sich die Fensterscheibe am unteren Anschlag befindet
- *MovingSensor_Front*: registriert die Fensterscheibenbewegung
- *TopFront*: registriert, ob sich die Fensterscheibe am oberen Anschlag befindet

- *Battery_Front*: registriert die aktuelle Batteriespannung
- *MotorFront*: registriert einen Befehl von der Komponente *Request_Front*

und den Ausgängen:

- *Front_Motor*: legt die Spannung an den Fensterhebermotor an
- *Front_Out_ERROR_WIN*: signalisiert einen Fehler
- *FrontLowWin*: signalisiert eine zu niedrige Batteriespannung

Die Komponente *FrontMotor* verhält sich so, wie in den Abschnitten 7.1.2 und 8.1.2 beschrieben wurde.

Mittels Modelchecking konnte für *FrontMotor* gemäss der TSG-Spezifikation folgende Eigenschaft nachgewiesen werden:
Für alle Systemzustände soll gelten: Wenn der Motor mindestens für die Länge der Startup-Zeit[7] *eingeschaltet ist und nach Ablauf dieser Zeit ein Einklemmen registriert wird, dann fahre die Fensterscheibe wieder runter, bis:*

- *der untere Anschlag erreicht worden ist,*
- *der 3-Sekunden-Timeout abgelaufen ist oder*
- *die Fensterbewegung blockiert wird (z.B. wegen einer mechanischen Fehlfunktion).*

Als LTL-Formel[8] ausgedrückt lautet diese Eigenschaft:

```
1.  [](
2.    (
3.      (Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up)
4.      && (()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up))
5.      && (()()()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up)))
6.      && (()()()()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up))))
7.      && (()()()()()()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up)))))
8.      && (()()()()()()()()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up)
9.                           && not(is_Msg(MC_FrontMotor.Top_Front))
10.                          && not(is_Msg(MC_FrontMotor.MovingSensor_Front))))))))
11.   ) => (
12.     (()()()()()()()()(
13.       ((
14.         Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down
15.       ) $Until (
16.         (not(is_Msg(MC_FrontMotor.Bottom_Front)) && not(is_Msg(MC_FrontMotor.MovingSensor_Front)))
17.         ||
18.         (Val(MC_FrontMotor.Bottom_Front)==F_UNTEN_Bottom)
19.       ))
20.       ||
21.       (
```

[7]Die Startup-Zeit ist die Zeit, die benötigt wird, um das Fenster in Bewegung zu setzen.

[8]Die Nummerierung am linken Rand dient nur zur Referenzierung.

```
22.         (Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)
23.         && (()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down))
24.         && (()(()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)))
25.         && (()(()(()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down))))
26.         && (()(()(()(()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)))))
27.         && (()(()(()(()(()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down))))))
28.         && (()(()(()(()(()(()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)))))))
29.         && (()(()(()(()(()(()(()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)))))))
30.         && (()(()(()(()(()(()(()(()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)))))))))
31.         && (()(()(()(()(()(()(()(()(()(Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)))))))))))
32.       )
33.     )))))))
34.   )
35. )
```

Im Einzelnen bedeuten die Zeilen:

Zeile 1: Für alle Systemzustände gelten Zeilen 2 bis 35.

Zeilen 3 bis 8: Das Ansteuern der Motoren erfolgt mindestens für 6 Systemticks (Abstraktion der Startup-Zeit).

Zeile 9 und 10: Das Fenster ist nach der Startup-Zeit noch nicht am oberen Anschlag und bewegt sich nicht.

Zeile 11: Wenn ein Einklemmen erkannt worden ist (Zeilen 3 bis 10), dann reagiere darauf (Zeilen 12 bis 24).

Zeile 12: Die Reaktion erfolgt zeitlich nach dem Einklemmen.

Zeilen 13 bis 19: Fahre die Schreibe runter (Zeile 14) bis (Zeile 15) die Scheibe blockiert (Zeile 16), oder schon am unteren Anschlag angelangt ist (Zeile 18),

Zeilen 22 bis 31: oder 10 Systemticks vergangen sind (Abstraktion des 3-Sekunden-Timeouts).

Die LTL-Formel greift dabei nicht auf Unterkomponenten und interne Kanäle von *FrontMotor* zurück, sondern nur auf die Kanäle der Schnittstelle nach aussen. Somit beschreibt die formulierte Eigenschaft das Verhalten von *FrontMotor* in Form einer Black-Box. Die Eingabe einer Nachricht in die Black-Box-Komponente (hier: *FrontMotor*) kann zu internen Berechnungsschritten führen. Dabei können Nachrichten intern eine Pipeline von Unterkomponenten durchlaufen. Es werden somit erst einige Systemticks benötigt, bis das Ergebnis am Ausgang vorliegt. Beim Aufstellen von Formeln in AutoFocus kommt dieser Pipelining-Effekt zum Vorschein: Soll eine gewünschte Eigenschaft gezeigt werden (z.B. „wenn X gilt, dann ergibt sich Y“), dann ist diese zeitliche Verzögerung zu berücksichtigen. Dieses zeitliche Vorausblicken äussert sich in einer Änderung der Formel („wenn X gilt, dann ergibt sich Y nach 2 System-

ticks“). Die resultierende Formel nimmt durch diese technischen Details ggf. eine unintuitive und aufgeblasene Form an. Zudem können kleine Verbesserungen des AutoFocus-Modells zu starken Änderungen an der Formel führen.

Von der anderen Seite her betrachtet, müsste man das AutoFocus-Modell anpassen, um eine einfache und klare Formel zu erhalten, die die gewünschte Eigenschaft ausdrückt. Ein Mittel dazu wäre das Entfernen der Nachrichten-Pipelines über „Immediate Ports“. Eine andere Möglichkeit ist das Zusammenfassen eines Komponentennetzwerkes zu einer SSD-Komponente. In der zugehörigen STD können dann mehrere sequentielle Berechnungsschritte unter Umständen zu einem Schritt zusammengefasst werden. Wie schon eben angedeutet, führt das Abändern eines AutoFocus-Modells jedoch wiederum zu einer anderen Formel.
Auf diese Weise bauen *FrontMotor* und obige LTL-Formel aufeinander auf (im Sinne von „Design for Modelcheckability“).

Dass die Einklemmschutzeigenschaft auch im Kontext des gesamten TSG-Modells korrekt ist, zeigt die folgende Überlegung.

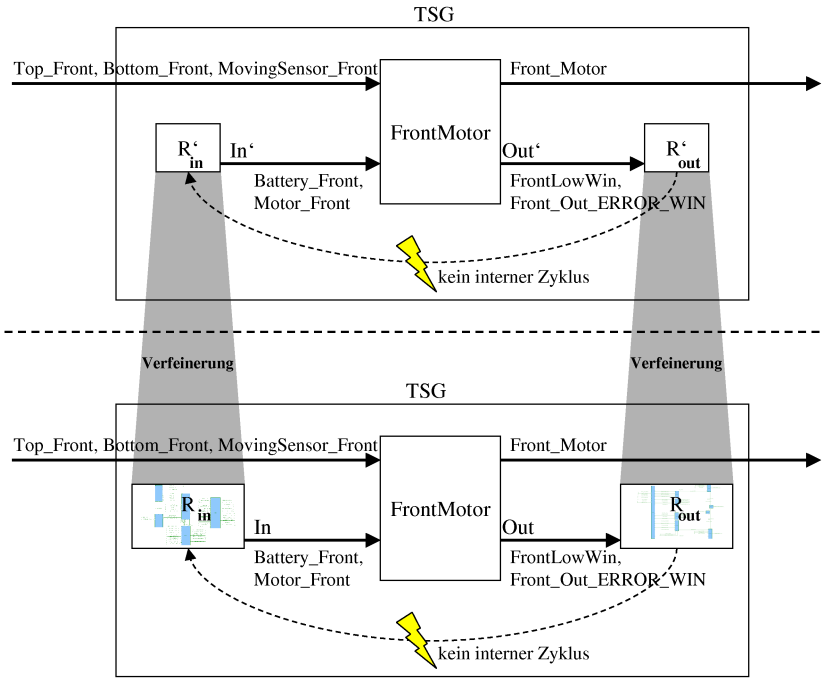


Abbildung 8.7: Modularer Verifikationsansatz.

Die Kanäle *Bottom_Front*, *MovingSensor_Front*, *Top_Front* und *Front_Motor* interagieren direkt mit der TSG-Umgebung. Sie sind somit nicht vom restlichen TSG-Modell abhängig. Die Eingangskanäle *Battery_Front* und *MotorFront* hingegen erhalten Nachrichten von anderen TSG-Komponenten. Nachrichten über die Ausgangskanäle *FrontLowWin* und *Front_Out_ERROR_WIN* gelangen ebenfalls nur über andere TSG-Komponenten an die TSG-Umgebung. Allerdings sind Ausgaben über *FrontLowWin* und *Front_Out_ERROR_WIN* nicht TSG-intern über *Battery_Front* und *MotorFront* rückgekoppelt. Dadurch ergeben sich für die Komponente *FrontMotor* keine internen Rückkoppelungen zwischen Teilkomponenten des TSGs.

In AutoFocus werden beim Abgehen des Zustandsraumes während des Modelcheckings alle Eingangskanäle der zu überprüfenden Komponente jeweils mit allen möglichen Werten belegt. Für die Komponente *FrontMotor* wird mit dieser „chaotischen“ Eingabemenge die Korrektheit obiger Eigenschaft nachgewiesen.
Im Kontext des TSG-Modells kann man daher die anderen TSG-Komponenten als zwei Komponenten R'$_{in}$ und R'$_{out}$ auffassen (Abstraktion). R'$_{in}$ liefert dann nichtdeterministisch Eingabewerte *In'* an *FrontMotor* über *Battery_Front* und *MotorFront*. R'$_{out}$ akzeptiert beliebige Ausgabewerte *Out'* von *FrontMotor* über *FrontLowWin* und *Front_Out_ERROR_WIN*.

Die Einbettung von *FrontMotor* in den Kontext des eigentlichen TSG-Modells, kann als Verfeinerung R$_x$ von R'$_x$ angesehen werden[9]. Denn die, von R$_{in}$ gelieferten Eingabedaten *In* (über *Battery_Front* und *MotorFront*) stellen eine Untermenge der, von R'$_{in}$ erzeugten chaotischen Eingabedatenmenge *In'* dar. Die Eingabe der Kanäle *Bottom_Front*, *MovingSensor_Front* und *Top_Front* bleibt weiterhin chaotisch, da diese Kanäle nach wie vor mit der externen TSG-Umgebung interagieren und nicht über R'$_{in}$ bzw. R$_{in}$ an *FrontMotor* gelangen. R$_{out}$ besteht aus den beiden STDs *ResolveError* und *ResolveBattery*, die die Ausgabe von *FrontMotor* nur nach aussen weiterleiten. Also ändert sich nichts an der akzeptierten Ausgabe über den Kanal *Front_Motor* (*Out'*=*Out*). Die Ausgabemenge von *FrontMotor* wird somit nicht durch die Verfeinerung eingeschränkt.

Wenn man diese Verfeinerung nun aus Sicht der Komponente *FrontMotor*

[9] „x“ steht für „in“ bzw. „out“.

betrachtet, so wird beim Übergang von R'$_x$ zu R$_x$ daher nur die Eingabemenge der Kanäle *Battery_Front* und *MotorFront* eingeschränkt. Wenn die Eigenschaft für alle möglichen Eingaben korrekt ist, dann trifft dies auch für einen Teil der Eingaben zu. Die Gültigkeit der obigen Eigenschaft bleibt damit im Kontext des Gesamtmodells R$_x$ erhalten.

Kapitel 9

Evaluierung

9.1 Bewertung der Werkzeuge

Beim folgenden Vergleich der beiden Werkzeuge AutoFocus und ASCET sollten einige Aspekte im Hinterkopf behalten werden:
Der Hintergrund, vor dem AutoFocus entwickelt wurde ist ein anderer, als bei ASCET. Der AutoFocus-Ansatz wurde möglichst generisch gewählt, um ein möglichst grosses Spektrum von Anwendungen zu unterstützen. ASCET hingegen war von Anfang an auf die Entwicklung von Steuergeräten und regelungstechnischen Algorithmen ausgerichtet und somit für den industriellen Einsatz im Bereich Automotive spezialisiert. Relativiert wird diese Situation durch die aktuell voranschreitende Entwicklung von AutoFocus 2, die momentan am Lehrstuhl von Professor Broy an der TU München vorangetrieben wird [Auta]. Vielversprechende innovative Konzepte zur Realisierung eines durchgängigen Entwicklungsprozesses werden darin integriert.

Daher konzentrieren sich die folgenden Abschnitte auf den Vergleich der dahinterliegenden Prinzipien und Methoden der beiden Werkzeuge. Es wird an dieser Stelle auch betont, dass einige Einschränkungen in AutoFocus nur durch das Modelchecking und nicht durch das Werkzeug selbst bedingt sind.

9.1.1 Modellierung

Die einfache intuitive Beschreibungstechnik in AutoFocus basiert auf einer eingeschränkten Notation im Vergleich zu anderen Modellierungswerkzeugen wie ASCET. Daraus resultiert eine kompakte dahinterliegende Semantik bei AutoFocus, die formal festgehalten werden kann und auch für andere akademische Verifikationsverfahren wie Theorem Proving zugänglich ist.

Ein weiterer sich daraus ergebender Vorteil ist die Allgemeinheit der Methodik. Einsatzgebiete von AutoFocus beschränken sich damit nicht auf eingebettete Systeme in der Automobilindustrie. Beispielsweise können ebenso Geldtransfersysteme zur Untersuchung von Sicherheitslücken modelliert werden [JW01].

Allerdings erschwert dieser allgemeine Ansatz im Gegenzug dazu eine tiefgehende Integration in bestehende Konzepte in der Automobilindustrie. Die in spezialisierten Werkzeugen (wie ASCET) integrierten Konzepte sind beispielsweise:

- Scheduling in OSEK mittels Tasks und Prozesse
- Abbildung der Datentypen auf die, von der Hardware unterstützten Datentypen
- Unterscheidung von Programm- und Datenstand

AutoFocus eignet sich somit vor allem für die Beschreibung der abstrakten Logik einer Funktion. Die Einbettung in den darunterliegen Hardware-Kontext und die Nachbildung der obigen Konzepte sollte jedoch nicht mehr im Rahmen der Modellierung in AutoFocus erfolgen. Technische Details einer Verfeinerung des Modells würden die Grösse und Komplexität des Modells (auch im Hinblick auf die Zustandsraumexplosion) stark anwachsen lassen. Problematisch dabei ist der Pipeline-Effekt (vgl. Abschnitt 8.1.2) und damit verbundene Synchronisationsprobleme, bedingt durch die parallele Berechnung des Komponentennetzwerkes.

In ASCET spielt die Synchronisation der einzelnen Komponenten ebenfalls eine Rolle. Allerdings ist hier eine sequentielle Abarbeitung durch den Prozessablauf und das Task-Scheduling vorgegeben (vgl. Abschnitt 6.1.1). Dies ist ein bedeutender Unterschied zum AutoFocus-Konzept mit parallel arbeitenden Komponenten, deren Berechnungen bei jedem

Systemtick synchron voranschreiten.

Auf der Ebene der Verhaltensbeschreibung von Komponenten setzt sich die Differenzierung fort: Die Auswahl von STD-Transitionen in AutoFocus erfolgt (bei gleicher Priorität und erfüllten Vorbedingungen) indeterministisch. Der Benutzer von AutoFocus muss darauf achten, dass die Auswahl der Transitionen durch entsprechende Vorbedingungen und Prioritätenvergabe deterministisch bleibt. Ein deterministischer Ablauf muss somit explizit erzwungen werden. In ASCET sind die Berechnungen in Zustandsautomaten hingegen deterministisch (unterschiedliche Transitions-Prioritäten sind obligatorisch).

Die Modellierung von zeitgesteuertem Verhalten in AutoFocus erfordert auf Grund des globalen Systemticks die Diskretisierung der Zeit. Timer-Komponenten können in AutoFocus mit Hilfe von Integer-Variablen realisiert werden. Die Auflösung eines Timeout-Intervalls wird dann durch eine Anzahl von Ticks bestimmt.
In ASCET-Modellen werden Zeitangaben direkt (ohne Diskretisierung) im Modell angegeben. Die Codegenerierung in ASCET-SE befasst sich dann mit der Abbildung auf entsprechende Hardware-Timer und deren korrekte Initialisierung.

Durch das Modelchecking ergeben sich in AutoFocus, bedingt durch die Zustandsraumexplosion, Einschränkungen: Eine Begrenzung der Schritte bzw. Integerwerte ist notwendig. Benutzerdefinierten Datentypen dürfen nicht rekursiv definiert werden.
ASCET unterstützt keine rekursiv definierten Datentypen auf der Modellebene.

AutoFocus ist in Kombination mit Modelchecking für ereignisgesteuerte Systeme ohne zeitliche Bedingungen prädestiniert.

9.1.2 Zeitlicher Aufwand bei der Modellierung

Einarbeitung:

Die, im Vergleich zu ASCET eingeschränkte Notation von AutoFocus kann innerhalb eines Tages erlernt werden.

ASCET hingegen bedurfte einer ungleich längeren Einarbeitungsphase von über einer Woche. Grund ist die Vielfalt von Ausdrucksmöglichkeiten

und die Erarbeitung der, im Bereich Automotive gängigen Konzepte.

In beiden Fällen mangelte es nicht an Dokumentation und Ansprechpartnern.

Modellierung in AutoFocus:

Die Modellierung der Komponenten *DoorLock* und *Windows* erfolgte in jeweils drei Wochen. *Light* wurde in einer Woche erstellt. Die Schwierigkeit lag in der korrekten Interpretation und Umsetzung der textuellen Spezifikation. Weiterhin hatten nachträgliche Korrekturen von logischen Fehlern und das Beheben von Synchronisationsproblemen aufwendige Umstrukturierungen zur Folge.

Beispiel 9.1.1 *Das gleichzeitige manuelle Öffnen des Beifahrerfensters durch die Fahrerfensterhebertaste und die Beifahrerfensterhebertaste dauert solange an, bis beide Tasten losgelassen wurden (falls zwischenzeitlich keine weiteren Ereignisse auftreten). Eine naive Modellierung würde z.B. alle Signaländerungen als neuen Befehl interpretieren und gemäss Spezifikation diesen dann auch gleich ausführen. Wenn nun der Fahrer seine Taste zuerst loslassen und der Beifahrer seine Taste noch gedrückt halten würde, dann würde erstere Aktion fälschlicherweise als Stopbefehl für die Fensterbewegung interpretiert werden. Eine Berücksichtigung dieses Umstandes führt zu einem komplexeren Modell.*

Der im Vergleich zu ASCET höhere Zeitaufwand resultierte unter anderem auch aus der restriktiven Notation und der fehlenden Möglichkeit zum Kopieren ganzer SSD-Hierarchien (inklusive dazugehörigen STDs). In AutoFocus 2 wurde dieses Defizit mittlerweile behoben. Dieser Aufwand entfällt, falls das Modell im Vornherein feststeht und sich nicht mehr im Nachhinein ändert. In der Praxis kann diese Annahme jedoch nicht getroffen werden, da sich Anforderungen auch während der Entwicklungphase ändern können. Relativiert wird dies durch die Tatsache, das die erwähnte Kopierfunktion in AutoFocus 2 zur Verfügung stehen wird.

Im Rahmen des Modelcheckings wurden weitere Umstrukturierungen in *Windows* vorgenommen, um das Problem der Zustandsexplosion in den Griff zu bekommen (vgl. Abschnitt 8.3.1). Dieser Eingriff dauerte eine Woche.

Modellierung in ASCET:

Dadurch, dass bei der Modellierung in AutoFocus die Hauptprobleme bei der Interpretation der TSG-Spezifikation schon blossgestellt waren, fiel die Umsetzung in ASCET leichter. Auch die liberaleren Beschreibungsmittel trugen dazu bei. Somit konnten *DoorState*, *Windows* und die Beleuchtung *DoorLight* bzw. *InteriorLight* in der halben Zeit wie bei AutoFocus fertiggestellt werden. Die prinzipielle Struktur wurde grösstenteils übernommen und in eine kompaktere Form gebracht. Die Anwendung der Testverfahren führte nur zu kleineren Änderungen.

Fairerweise muss erwähnt werden, dass die Umsetzung und Interpretation der Spezifikation fast die Hälfte der Zeit für die Bearbeitung in AutoFocus ausgemacht hat. Bei der Modellierung in ASCET fiel diese Zeit zum Grossteil nicht mehr an, da auf die vorher in AutoFocus gesammelte Erfahrung zurückgegriffen werden konnte. Das grundlegende Lösungskonzept wurde schon in AutoFocus erarbeitet. Somit kann bei beiden Tools von einem äquivalenten Zeitaufwand für die Modellierung ausgegangen werden, wenn man von den nachträglichen Umstrukturierungen in AutoFocus im Rahmen des Modelcheckings absieht.

Tabelle 9.1 stellt den zeitlichen Aufwand für die Modellierung in beiden Tools nochmals gegenüber.

	ASCET[1]	**AutoFocus und Spezifikation**
Einarbeitung	1 bis 2 Wochen	1 Tag
Modellierung: DoorLock	1,5 Wochen	3 Wochen
Modellierung: Windows	1,5 Wochen	3 Wochen
Modellierung: Light	0,5 Wochen	1 Woche

Tabelle 9.1: Zeitlicher Modellierungsaufwand.

[1] In AutoFocus wurde hierbei schon Vorarbeit geleistet (siehe Text).

9.1.3 Zusammenhang zwischen Modell und Code

Die Umsetzung der Modelle in C-Code erfolgt in AutoFocus und ASCET auf eine kanonische Weise. Beispielsweise wird in beiden Tools für einen Zustandsautomaten jeweils eine C-Funktion erzeugt. Sie besteht im Grossen und Ganzen aus einem einzigen switch-Statement, das den aktuellen Zustand des Automaten prüft. Bei jedem Aufruf der Funktion wird das switch-Statement einmal durchlaufen. In AutoFocus geschieht dies bei jedem Systemtick und in ASCET in jeder Scheduling-Runde.

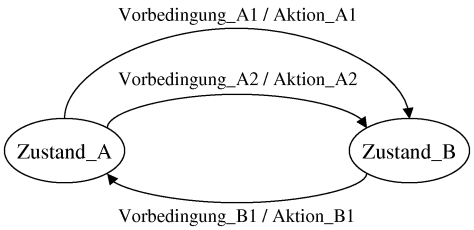


Abbildung 9.1: Beispielautomat zur Demonstration der Codeerzeugung.

Pseudocode zum Automaten von Bild 9.1:

```
void Funktionsberechnung() {
  switch(aktuellerZustand) {
    case(Zustand_A) : {
      if(Vorbedingung_A1) {
        Aktion_A1();
        aktuellerZustand = Zustand_B;
      } else if(Vorbedingung_A2) {
        Aktion_A2();
        aktuellerZustand = Zustand_B;
      }
    }
    case(Zustand_B) : {
      if(Vorbedingung_B1) {
        Aktion_B1();
        aktuellerZustand = Zustand_A;
      }
    }
  }
```

```
}
```

ASCET erlaubt dem Benutzer zusätzlich noch das Hinzufügen von implementationsabhängigem C-Code (vgl. Bild 9.2). Über Makros kann darin auf die Modellelemente zugegriffen werden.

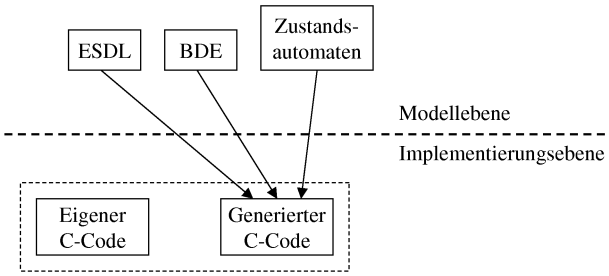


Abbildung 9.2: Code-Einbindung in ASCET.

Im Vergleich zur Entwicklung mit den beiden Werkzeugen bietet die Programmierung auf Codeebene einen grösseren Freiheitsgrad hinsichtlich der Beschreibungsmöglichkeiten. Vor allem bei hardwarenahen Funktionen wird dies deutlich. Beispielsweise unterstützt ASCET-RP 5.1 in Verbindung mit der Rapid-Prototyping-Karte ES 1222 keine Versendung von azyklischen CAN-Nachrichten im direct mode mit Hilfe von Modellelementen und Standardeinstellungen. ASCET kompensiert dies über Workarounds durch die Möglichkeit, die Funktion mit benutzerspezifischem C-Code nachzubilden.
AutoFocus hat vom Umfang her restriktivere Beschreibungstechniken und damit geringere Freiheitsgrade. Letzteres ist allerdings darauf zurückzuführen, dass AutoFocus aus der Idee heraus entstand, eingeschränkte Beschreibungsmittel zur Verfügug zu stellen, die im Gegenzug dazu dann mit formalen Methoden behandelt werden können (vgl. Abschnitt 9.1.1).
Hardwarenahe Codeabschnitte können in AutoFocus nach der Codegenerierung von Hand eingefügt werden. Allerdings muss dieser Vorgang bei jeder Abänderung des Modells und nachfolgender Codegenerierung wiederholt werden.
Eine automatisierte rückwirkende Änderung des Modells nach einer Codeänderung ist sowohl bei AutoFocus als auch bei ASCET momentan nicht gegeben. Ohne Einschränkungen bei der Programmierung mittels Richtlinien ist solch ein Ansatz praktisch auch nur schwer realisierbar.

Grund ist der grosse Freiheitsgrad seitens C. Eine direkte bijektive Abbildung einzelner C-Konstrukte nach AutoFocus würde zudem den Vorteil der Modellierung zunichte machen. Man könnte dann gleich in C programmieren.

Vorteile der modellbasierten Entwicklung gegenüber der Programmierung auf Codeebene ergeben sich durch die bessere Verständlichkeit. Zusammenhänge können grafisch visualisiert und damit schneller erkannt werden. Bei steigender Team- und Projektgrösse lässt sich der Überblick leichter behalten. Die hierarchische Strukturierung des Systems trägt dazu bei. Auch Laien können sich somit schneller in ein existierendes Modell einarbeiten, was die spätere Wartung eines Systems durch Dritte vereinfacht. Weiterhin können Standardkomponenten Bibliotheken entnommen werden und per drag & drop einem System hinzugefügt werden. Systeme, wie z.B. Regelstrecken lassen sich damit einfach zusammenklicken und „auf die Schnelle“ simulieren.

Allerdings können sich auch bei der Erstellung von Modellen Fehler einschleichen. Als präventive Massnahme können z.B. unternehmensspezifische Modellierungsrichtlinien (wie z.B. [Hua04]) dazu dienen, Leichtsinnsfehler zu vermeiden. Auch deren automatische Überprüfung durch entsprechende Werkzeuge ist möglich.
Trotz der, in Abschnitt 2.2 genannten Vorteile der automatischen Codegenerierung, ist der Einsatz in manchen Fällen nicht möglich. Beispielsweise wenn hochoptimierter Code für den Einsatz in zeitkritischen oder speicherarmen Umgebungen nötig ist.
Weiterhin gibt es kein Werkzeug, das alle Einsatzgebiete abdeckt und zugleich einen effizienten durchgängigen Entwicklungsprozess unterstützt. ASCET ist beispielsweise auf letzteres im Bereich der Entwicklung für Steuergeräte-Software und regelungstechnische Algorithmen ausgerichtet. Multimedia-Anwendungen oder Anwendungen abseits des Bereichs Automotive werden hingegen nicht betrachtet.

9.2 Bewertung der Qualitätssicherungsmethoden

Fehlerfindung:

Unstimmigkeiten mit der TSG-Spezifikation wurden in ASCET mit „Offline-Experimenten“ ausgemacht (siehe Abschnitt 7.2). Unter den gefundenen Fehlern befanden sich z.B. Deadlocks oder fehlerhafte Transitionsvorbedingungen, die zu einer falschen Priorisierung geführt haben. Die Tests liefen Hand in Hand mit der Modellierung der einzelnen Komponenten. Das fertige ASCET-Modell wurde im Nachhinein modulweise mit der in Abschnitt 7.3 vorgestellten Methode auf Codeebene getestet. Es wurden hierbei kleinere Leichtsinns-Fehler (z.B. Übertragung eines falschen Nachrichtenwertes), nicht erreichbare Stellen im Modell und falsche Ablaufsequenzen (in ASCET) gefunden.

In AutoFocus war der Simulator recht hilfreich, um unmittelbar nach dem Erstellen einer Teilkomponente logische Fehler (z.B. falsche Priorisierung von Transitionen) zu entlarven. Schwer auffindbare Fehler, die vor allem die Synchronisation und Randbedingungen betrafen, wurden mit dem Modelchecker gefunden. Beispielsweise wurde so der verzögerte Timeout-Alarm in Abschnitt 8.1.2 ausgemacht. Ohne den Einsatz des Modelcheckers wären wesentlich mehr Testläufe notwendig gewesen, um Fehler dieser Art ausfindig zu machen.

Tabelle 9.2 stellt die Anzahl der gefundenen Fehler in beiden Tools gegenüber. Es werden hier nur Fehler betrachtet, die erst nach der Fertigstellung der Komponenten aufgedeckt worden sind und nicht schon während des schrittweisen Aufbaus der Komponenten. Bei den Angaben muss beachtet werden, dass dem Modelchecking und White-Box-Testen jeweils die Simulation bzw. das Experiment vorausging. Der Grossteil der Fehler wurde somit schon vor Beginn des Modelcheckings bzw. White-Box-Testens abgefangen.

	ASCET	**Auto**Focus
Experiment / Simulation	10	5
Modelchecking	-	5
White-Box-Testen	5	-

Tabelle 9.2: Anzahl gefundener Fehler.

Intellektueller Aufwand:

Der Hauptaufwand beim Testen in ASCET bestand vor allem in der

Interpretation der Testausgaben und dem Abgleich mit den Anforderungen der Spezifikation. Im Allgemeinen lässt sich der Testaufwand mit wachsendem Automatisierungsgrad der Tests reduzieren. So existieren auch Ansätze, die Anforderungen mit der Testausführung zu koppeln. Tools, wie DOORS von Telelogic stehen dafür zur Verfügung ([Tel]). Die Schwierigkeit besteht dabei in der Erfassung von komplexen Zusammenhängen und Abhängigkeiten. Das Vorliegen einer Spezifikation auf formaler Basis würde diesen Prozess sehr unterstützen. Im Rahmen der Diplomarbeit musste jedoch auf eine Spezifikation in textueller Prosaform zurückgegriffen werden. Schritte mit Interpretationstätigkeiten (z.B. Vergleichen der Ausgabe mit den Anforderungen in der Spezifikation) liessen sich damit nicht maschinell durchführen und erwiesen sich als langwieriger Prozess.

Modelchecking in AutoFocus wurde zunächst eingesetzt, um Teilkomponenten auf lokale Eigenschaften hin zu überprüfen. Da Kenntnisse über den internen Aufbau der Komponenten vorlagen (White-Box), konnten mit geringem Aufwand während der Entwicklung gezielt Eigenschaften (z.B. Abwesenheit von Deadlocks) formuliert und überprüft werden. Ein Fehler deutete entweder auf eine inkorrekte Stelle im Modell oder auf eine bzgl. der Spezifikation fehlerhaft formulierte Eigenschaft hin. Die Schwierigkeit bei der Überprüfung mittels Modelchecking besteht zunächst im Aufstellen korrekter Eigenschaften, die im Einklang mit der Spezifikation stehen. Gezwungenermassen setzt man sich dabei detaillierter mit der Spezifikation auseinander und deckt dadurch unter Umständen Spezifikationsfehler auf.

Das nachträgliche Aufstellen von Eigenschaften ohne Berücksichtigung der internen Struktur der Komponenten (Black-Box) oder von Komponenten einer höheren Hierarchiebene gestaltet sich schwierig (vgl. Abschnitt 8.3.1). Beispielsweise können durch die Zeitdiskretisierung zeitliche Abstände mittels Verkettung von Next-Operatoren ausgedrückt werden. Nicht immer lassen sich damit auf Grund von Wechselwirkungen Eigenschaften auf eine intuitive Art und Weise darstellen. Die verzögerte Nachrichtenübermittlung in AutoFocus bei Verzicht auf „immediate ports“ führt zu komplexeren Formeln, um die zeitlichen Abhängigkeiten zu erfassen. Mit wachsender Komplexität der Formel wird es auch zunehmends schwerer die Korrektheit der Formulierung der Eigenschaft selbst im Hinblick auf die Spezifikation zu zeigen.

Subjektiv betrachtet stellte dies jedoch, im Gegensatz zu den manuellen, sysiphusartigen Schritten beim Testen, eine herausfordernde Aufgabe dar. Auch die konstruktive Art, Eigenschaften nachzuweisen, statt Fehler finden zu müssen, wirkten sich positiv auf die Motivation aus.

Zeitlicher Aufwand:

Fast die gesamte Zeit für das Testen wurde zu gleichen Teilen für das Aufstellen von Testfällen (hinsichtlich eines hohen Überdeckungsgrades) und die Interpretation der Testlaufergebnisse aufgewandt. Wie schon oben erwähnt, könnte dieser Teil durch eine entsprechende Automatisierung hinsichtlich der Anforderungen reduziert werden. Die Rechenausführungszeit der Tests selbst (gemäss 7.3) kann in ASCET skaliert werden, so dass der dafür betriebene Aufwand praktisch irrelevant klein ist: Das Zeitintervall zwischen zwei Taskaktivierungen kann bei einem Offline-Experiment in ASCET durch den Benutzer variiert werden.

Beim Modelchecking wurde fast die gesamte Zeit für das Erstellen von Eigenschaften in Anspruch genommen. Diese mussten mit der Spezifikation und dem Modell abgeglichen werden (siehe oben). Die Ausführungszeit des Modelcheckers auf den TSG-Komponenten der unteren Hierarchieebenen betrug in diesem Fall im Durchschnitt unter 5 Minuten. Voraussetzung ist ein Modell mit kleinem Zustandsraum (vgl. Abschnitt 8.3). Im Allgemeinen kann jedoch nicht von solch einer relativ kurzen Zeit ausgegangen werden. Je nach Modellgrösse können dafür mehrere Stunden oder Tage aufgewandt werden. Es muss dann abhängig von der Sicherheitsrelevanz ein individueller Kompromiss zwischen Modellgenauigkeit und Zeitaufwand gefunden werden. Bei iterativen kleinen Verbesserungen und anschliessendem Modelchecking summieren sich diese Zeiten zudem auf und ziehen den Prozess in die Länge. Allerdings ist der Aufwand durch die Qualität der damit erzielten Ergebnisse (=Ausmachen von schwer zu findenden Fehlern) gerechtfertigt.

Wurde die Existenz eines Fehlers im Modell gezeigt, so nahm auch dessen Lokalisierung Zeit in Anspruch. Ein Gegenbeispiel muss zunächst ausgeführt und interpretiert werden. Auch dies erfordert ein intuitives Vorgehen im Zusammenhang mit der Spezifikation, was schwer zu automatisieren ist.
Diesen Prozess kann man durch Bounded Model Checking beschleunigen. Neben der Ausführungszeit des Modelcheckers reduziert sich dadurch auch die Länge der Gegenbeispiele.

In Tabelle 9.3 wird die aufgebrachte Zeit mit der dabei erreichten Fehleranzahl in Beziehung gesetzt. Zu beachten ist hierbei, dass jeweils in AutoFocus und ASCET der Überdeckungsgradanalyse bzw. dem Modelchecking die Simulation vorausging.

Verfahren	**Zeitaufwand (Stunden)**	**Fehleranzahl**
Experiment (ASCET)	24	10
Simulation (AutoFocus)	24	5
Modelchecking	80	5
White-Box-Testen	56	5

Tabelle 9.3: Gefundene Fehler je Zeitaufwand.

Die Interpretation der Zahlen besagt, dass im Rahmen dieser Fallstudie beim Modelchecking in derselben Zeit wie beim Coverage-Testen weniger Fehler gefunden wurden. Allerdings muss hinzugefügt werden, dass die dabei gefundenen Fehler beim Modelchecking schwerwiegendere Logik- und Synchronisationsfehler waren (siehe obiger Abschnitt "Fehlerfindung").

Weiterhin lässt sich der Unterschied durch die Natur des Modelcheckings begründen: Beim Modelchecking erfolgt normalerweise der konstruktive Nachweis von Eigenschaften. Das Aufdecken von Fehlern ergibt sich als Nebenprodukt in Form eines Gegenbeispiels. Im Gegensatz dazu ist Testen auf das Finden von Fehlern ausgerichtet.

Effizienz:

Beim Testen gilt die Devise: Je mehr Fehler gefunden wurden, desto erfolgreicher waren die Tests. Die Wahrscheinlichkeit, Fehler zu finden erhöht sich mit der Anzahl und Aussagekräftigkeit der Testläufe. Viele, grosse Testfälle im CTE-Tool führen zum Beispiel zu grossen Matrizen, was auch nicht gerade zur Übersichtlichkeit bei der Interpretation beiträgt. Ein automatisiertes Vorgehen ohne menschliche Eingriffe basierend und ausgehend von einer formalen Spezifikation wie bei der modelbasierten Testfallgenerierung könnte hier Abhilfe schaffen. Allerdings können, wie schon in Abschnitt 3.2.2 angedeutet, alle Pfade durch ein Programm im Allgemeinen praktisch nicht zu 100% ausgetestet werden.

Die Anzahl der möglichen Testfälle ist dazu zu hoch und die zeitlichen Ressourcen bei der Testplanung beschränkt.

Ähnlich verhält es sich beim Modelchecking. Man kann im Allgemeinen auch hier kein 100%-ig korrektes Modell unter allen Umständen nachweisen. Lediglich die Korrektheit der gestellten Fragen (hier: der formulierten Eigenschaften) kann nachgewiesen werden. Je mehr Eigenschaften nachgewiesen werden, desto grösser wird der Teil des abgedeckten Modellverhaltens. Im Idealfall wird für jede Anforderung der Spezifikation die entsprechende Eigenschaft nachgewiesen.

Folglich ist die Suche nach Fehlern bei komplexen Systemen einfacher als der Nachweis der Fehlerfreiheit eines Produkts. Trotz eines Korrektheitsnachweises nehmen folgende Fehlerquellen Einfluss auf das resultierende Produkt:

- Nichterfüllung der gestellten Annahmen (z.B. bei Hardware-Problemen)
- Fehlerhafte Spezifikation (bzw. fehlerhafte Interpretation einer textuellen Spezifikation)
- Inkorrektes Beweissystem (z.B. auf Grund von Programmierfehlern)

Einsatz von Modelchecking:

Entgegen der in Abschnitt 3.2 gestellten Forderung nach einer unabhängigen Testgruppe, wurde bei der Durchführung der Diplomarbeit aus pragmatischen Gründen darauf verzichtet. D.h., Entwicklung und Tests wurden hier von derselben Person erbracht. Dadurch ergab sich als zusätzlicher Gesichtspunkt die Anwendung von Modelchecking während des Entwicklungsprozesses (aus der Entwicklersicht), statt als unabhängiger Verifikationsprozess (aus Testersicht). Als hilfreiches Feature von AutoFocus konnten dabei Gegenbeispiele in Form von EETs über das TDF-Zwischenformat in den Simulator geladen und ausgeführt werden. Gefragte Eigenschaften betrafen vor allem die Erreichbarkeit bzw. den Ausschluss von Systemzuständen.

Grosse Komponenten konnten in AutoFocus nicht per Modelchecking überprüft werden (vgl. Abschnitt 8.3.1). Man könnte hier kompositionelles Modelchecking anwenden, was jedoch tiefgreifende Kenntnisse aus der Theorie voraussetzt und somit nicht mehr eine Push-Button-

Verifikationsmethode darstellt.
In der Praxis ist die Anwendung von Modelchecking durch den Entwickler nicht für komplexe Projekte geeignet, da dieser selbst mit vielen anderen Problemen aus der Anwendungsdomäne (z.B. Regelkreisstabilität, etc.) konfrontiert ist. Eine aufgezwungene Vorgehensweise und Restriktionen bei der Modellierung würde seine Freiheitsgrade und somit seine Effizienz einschränken. Daher sollten, entsprechend der Forderung nach einer separaten Testgruppe, Verifikationsspezialisten eingesetzt werden, die den Testprozess begleiten.

Kapitel 10

Fazit und Ausblick

Ein entscheidendes Defizit der Modelchecking-Methode ist das Problem der Zustandsraumexplosion. Befindet sich die Entwicklung jedoch noch in einem frühen Stadium bzw. liegt ein event-gesteuertes System auf einer höheren Abstraktionsebene vor, so sind die Bedingungen für das Modelchecking gegeben. Damit kann mit Hilfe von Verifikationstechniken die Grundlage für die folgenden Entwicklungsschritte und Testverfahren geschaffen werden. Beispielsweise können dann, basierend auf verifizierten Modellen, Testfälle für Back-to-Back-Tests erstellt werden.

Folgende Tabelle stellt allgemeine Gesichtspunkte der beiden Qualitätssicherungsmethoden nochmals gegenüber:

Testen	**Modelchecking**
untersucht ein physikalisches bzw. konkretes System	untersucht ein abstraktes Modell
In-The-Loop-Tests erfolgen in einer Umgebung, die dem realen Umfeld nahe kommt	billige und frühzeitige Verifikation (ohne komplexe In-The-Loop-Testumgebung)
kein Nachweis der Korrektheit von Eigenschaften möglich	Nachweis der Korrektheit von gefragten Eigenschaften möglich
verwendet oft viele, oberflächliche Testdaten (z.B. bei automatisierten Tests)	ausgewählte, benutzerspezifische Eigenschaften

Tabelle 10.1: Testen vs. Modelchecking.

Darüber hinaus existieren noch weitere vielversprechende Ansätze, Tools und Techniken zur Qualitätssicherung, auf die im Folgenden nur hingewiesen wird:

- Der „Symbolic Polyspace Verifier“ der Firma Polyspace verwendet das statische Verfahren der „abstrakten Interpretation“, um

Schwachstellen in einem Programm auszumachen [Pol].

- Einen weiteren Testansatz stellt das „evolutionäre Testen“ dar [sys]. Dabei werden Testfälle durch iterative Bewertung und Rekombination optimiert.
- Den, in AutoFocus betrachteten Systeme liegt ein globaler Systemtick und somit eine Diskretisierung der Zeit zu Grunde. Systeme können jedoch auch mit Hilfe von Zeitautomaten beschrieben werden. Somit lassen sich Echtzeitsysteme in Werkzeugen, wie Uppaal2k, modellieren und überprüfen [UPP].

Die hier angesprochenen und in den vorhergehenden Kapiteln behandelten Software-Qualitätssicherungsmethoden haben ein gemeinsames Ziel: Die Erstellung korrekt funktionierender Software.
Bekannt ist aber auch, dass eine Kette nur so stark ist, wie ihr schwächstes Glied. Daher sollte die Absicherung von Software im Einklang mit der Hardware stehen. Denn nur wenn diese die Annahmen erfüllt, die z.B. beim Modelchecking an das Modell gestellt wurden, ist ein fehlerfreier Ablauf des Gesamtsystems möglich. Ansätze wie HIL-Testverfahren zielen auf solch ein Zusammenspiel von Soft- und Hardware ab. Im Bereich Automotive sollte somit der Einsatz von formalen Verifikationstechniken kein Verzicht auf konventionelle Testverfahren bedeuten. Deutlich wird aber auch, dass Testverfahren und formale Verifikationtechniken komplementäre Qualitätssicherungstechniken darstellen, die sich gegenseitig ergänzen und in Kombination angewandt einen Synergieeffekt hervorbringen können. Weder die eine, noch die andere Technik können als die in allen Belangen Bessere identifiziert werden.

Das Potential von formalen Verifikationstechniken, wie Modelchecking, ist noch längst nicht ausgereizt, so dass zur Zeit auf diesem Gebiet eine gewisse Dynamik bei den Werkzeugherstellern herrscht. Dies rührt auch daher, dass die modellbasierte Entwicklung in industriellen Anwendungen immer mehr Fuss fasst und dabei ebenfalls in Bereiche der Hardware-Entwicklung vordringt. Ebenso lassen sich die Qualitätssicherungsverfahren nicht nur im Kontext der Automobilindustrie anwenden. Technische Systeme jeglicher Art, deren Software gewissen sicherheitskritischen Anforderungen Stand halten muss, können aus dem Einsatz von Modelchecking analog zu bisherigen Betrachtungen Vorteile ziehen.

Einen wichtigen Meilenstein in der Automobilindustrie wird der aufkommende AUTOSAR-Standard darstellen. Die Schnittstelle zwischen den einzelnen Software- und Hardware-Schichten kann als Grundlage für Annahmen dienen, die z.B. beim Modelchecking getroffen werden können. Verifikationstechniken können somit ebenso von diesem modularen Vorgehen profitieren.

Werkzeuge, die das AUTOSAR-Konzept unterstützen, werden in der Automobilindustrie Zukunft haben. Man darf also gespannt sein, wie sich die Lage weiterentwickeln wird.

Anhang A

Spezifikationssprachen

A.1 CTL

A.1.1 Syntax

CTL-Formeln ergeben sich durch die rekursive Anwendung der Regel

p ::= P | (p $\vee$ p) | $\neg$ p | AXp | EXp | AFp | EFp | AGp | EGp | A(p U p) | E(p U p)

mit P als Aussagenvariable

A.1.2 Semantik

Die folgenden Beziehungen gelten für den Abwicklungsbaum T = $t(\mathcal{M}, s)$ einer Kripke-Struktur $(\mathcal{M}, s)$ mit M = (S,R,L) über den Aussagenvariablen $p_1, ...p_n$. φ und ψ stellen dabei CTL-Formeln dar. Für eine Folge $\pi = s_0 s_1 ... s_i s_{i+1} ... s_n$ bezeichnet der Ausdruck $\pi[i]$ das Folgenglied s_i.

$T \vDash p_i \iff p_i \in L(s)$
$T \vDash \neg\varphi \iff T \nvDash \varphi$
$T \vDash \varphi \vee \psi \iff T \vDash \varphi$ oder $T \vDash \psi$
$T \vDash EX\varphi \iff$ Es gibt einen Nachfolger s′ von s mit $t(\mathcal{M}, s') \vDash \varphi$
$T \vDash AX\varphi \iff$ Für alle Nachfolger s′ von s gilt $t(\mathcal{M}, s') \vDash \varphi$
$T \vDash EF\varphi \iff$ Es gibt einen Pfad π in T mit $\exists i \in \mathbb{N}. t(\mathcal{M}, \pi[i]) \vDash \varphi$

$T \vDash EG\varphi \iff$ Es gibt einen Pfad π in T mit $\forall i \in \mathbb{N}.t(\mathcal{M}, \pi[i]) \vDash \varphi$
$T \vDash AF\varphi \iff$ Für alle Pfade π in T gilt $\exists i \in \mathbb{N}.t(\mathcal{M}, \pi[i]) \vDash \varphi$
$T \vDash AG\varphi \iff$ Für alle Pfade π in T gilt $\forall i \in \mathbb{N}.t(\mathcal{M}, \pi[i]) \vDash \varphi$
$T \vDash E(\varphi U \psi) \iff$ Es existiert ein Pfad π in T und ein $i \in \mathbb{N}$, so dass $t(\mathcal{M}, \pi[i]) \vDash \psi$ und $\forall j < i.t(\mathcal{M}, \pi[i]) \vDash \varphi$
$T \vDash A(\varphi U \psi) \iff$ Für alle Pfade π in T existiert ein $i \in \mathbb{N}$, so dass $t(\mathcal{M}, \pi[i]) \vDash \psi$ und $\forall j < i.t(\mathcal{M}, \pi[i]) \vDash \varphi$

A.2 LTL

A.2.1 Syntax

LTL-Formeln ergeben sich durch die rekursive Anwendung der Regel

p ::= P | (p $\vee$ p) | $\neg$ p | Xp | Fp | Gp | pUp | pRp

mit P als Aussagenvariable

A.2.2 Semantik

Die folgenden Bemerkungen gelten für den Abwicklungsbaum T = $t(\mathcal{M}, s)$ einer Kripke-Struktur $(\mathcal{M}, s)$ mit M = (S,R,L) über den Aussagenvariablen $p_1, ...p_n$. φ und ψ stellen hier LTL-Formeln dar. LTL-Formeln werden über Pfade π interpretiert. Für einen Pfad $\pi = s_0 s_1 ... s_i s_{i+1} ... s_n$ bezeichnet der Ausdruck $\pi[i]$ das Pfadelement s_i. $\pi \uparrow$i bezeichnet den Pfad, der ab der Position i im ursprünglichen Pfad π beginnt, wobei ab 0 indiziert wird. Die Menge der natürlichen Zahlen $\mathbb{N}$ enthalte hier die 0. Eine Kripke-Struktur $(\mathcal{M}, s)$ erfüllt eine LTL-Formel φ gdw. alle (unendlichen Pfade) von $t(\mathcal{M}, s)$ die Formel φ erfüllen. Folgende Beziehungen drücken aus, wann ein Pfad π eine Formel erfüllt.

$\pi \vDash p_i \iff p_i \in L(\pi[0])$
$\pi \vDash \neg\varphi \iff \pi \nvDash \varphi$
$\pi \vDash \varphi \vee \psi \iff \pi \vDash \varphi$ oder $\pi \vDash \psi$
$\pi \vDash X\varphi \iff \pi \uparrow 1 \vDash \varphi$
$\pi \vDash G\varphi \iff \forall i \in \mathbb{N}.\pi \uparrow i \vDash \varphi$
$\pi \vDash F\varphi \iff \exists i \in \mathbb{N}.\pi \uparrow i \vDash \varphi$

$$\pi \vDash \varphi U \psi \qquad \Longleftrightarrow \exists i \in \mathbb{N}.[(\pi \uparrow i \vDash \psi) \wedge \forall j.(1 \leqq j < i \Rightarrow \pi \uparrow j \vDash \varphi)]$$

Anhang B

Hardware-Schnittstelle des Türsteuergerätes

Folgende Tabellen geben eine Übersicht über die Zuordnung der Anschlüsse der peripheren Komponenten zu den Steckerschnittstellen des Türsteuergeräts. Weitere Details können [PH02] entnommen werden.

B.1 Stecker S1: Anschluss von Elementen der Vordertür

Bezeichnung	**Richtung**	**Bedeutung**
FHB_VL	Input	Fensterheber-Taste vorne links (nur auf Fahrerseite belegt)
FHB_VR	Input	Fensterheber-Taste vorne rechts (nur auf Fahrerseite belegt)
FHB_HL	Input	Fensterheber-Taste hinten links (nur auf Fahrerseite belegt)
FHB_HR	Input	Fensterheber-Taste hinten rechts (nur auf Fahrerseite belegt)
F_BEWEG	Input	Bewegungssensor für die Vorderscheibe
T_OFFEN	Input	Zustandssensor der Vordertür
T_GRIFF	Input	Zustandssensor des Türgriffs der Vordertür

T_LICHT	Output	Ansteuerung der Ausstiegsleuchte der Vordertür
F_UNTEN	Input	Positionssensor für vordere Fensterscheibe unten
F_OBEN	Input	Positionssensor für vordere Fensterscheibe oben
T_RIEGEL	Input	Zustandsensor für vordere Türverriegelung
KEY_STATE	Input	Zustandssensor des Türschlosses
ZENTR_MOT1	Output	1. Anschluss für den Verriegelungsmotor der Vordertür
ZENTR_MOT2	Output	2. Anschluss für den Verriegelungsmotor der Vordertür
F_MOTOR1	Output	1. Anschluss für den Fensterhebermotor der Vorderscheibe
F_MOTOR2	Output	2. Anschluss für den Fensterhebermotor der Vorderscheibe

B.2 Stecker S2: Anschluss von externen Elementen

Bezeichnung	**Richtung**	**Bedeutung**
FFHB	Input	Fensterhebertaste der Hintertür
FF_BEWEG	Input	Bewegungssensor für die Hinterscheibe
FF_MOTOR1	Output	1. Anschluss für den Fensterhebermotor der Hinterscheibe
FF_MOTOR2	Output	2. Anschluss für den Fensterhebermotor der Hinterscheibe
FF_UNTEN	Input	Positionssensor für hintere Fensterscheibe unten
FF_OBEN	Input	Positionssensor für hintere Fensterscheibe oben
FT_OFFEN	Input	Zustandssensor der Hintertür

FT_GRIFF	Input	Zustandssensor des Türgriffs der Hintertür
FKIND_SICH	Input	Zustand der Kindersicherung der Hintertür
FT_LICHT	Output	Ansteuerung der Ausstiegsleuchte der Hintertür
FT_RIEGEL	Input	Zustandssensor für hintere Türverriegelung
FZENTR_MOT1	Output	1. Anschluss für den Verriegelungsmotor der Hintertür
FZENTR_MOT2	Output	2. Anschluss für den Verriegelungsmotor der Hintertür

Anhang C

Modellierung in ASCET

C.1 Benutzerdefinierte Datentypen

Datentyp	**Wert**	**Bedeutung**
AUTO_CLOSE	AUTO_CLOSE_Disabled	TSG-Konfigurationswert: keine Wirkung
	AUTO_CLOSE_Roof	TSG-Konfigurationswert: Schliessen der Fenster beim Abschliessen, wenn Cabriolet-Verdeck geschlossen ist
	AUTO_CLOSE_Always	TSG-Konfigurationswert: Beim Abschliessen werden Fenster immer geschlossen
BATT	BATT_ge0lt90	$0V \leq$ Batteriespannung $< 90V$
	BATT_ge90lt100	$90 \leq$ Batteriespannung $< 100V$
	BATT_ge100le105	$100 \leq$ Batteriespannung $\leq 105V$
	BATT_gt105le180	$105 <$ Batteriespannung $\leq 180V$
BLINK	BLINK_Off	kein Blinken
	BLINK_50	0,5 Sekunden lang Blinken
	BLINK_100	1 Sekunde lang Blinken
FHB	FHB_Neutral	kein Befehl von der Fensterheberbedienung
	FHB_DownAuto	Befehl zum automatischen Herunterfahren des Fensters bis zum Anschlag
	FHB_DownMan	Befehl zum Herunterfahren des Fensters
	FHB_UpMan	Befehl zum Hochfahren des Fensters
	FHB_UpAuto	Befehl zum automatischen Hochfahren des Fensters bis zum Anschlag
FZENTR_MOT	FZENTR_MOT_Neutral	kein Befehl an Fond-Verriegelungsmotoren
	FZENTR_MOT_Open	Befehl zum Öffnen an Fond-Verriegelungsmotoren
	FZENTR_MOT_Close	Befehl zum Schliessen an Fond-Verriegelungsmotoren
KEY_STATE	KEY_STATE_Neutral	Türschloss nicht gedreht
	KEY_STATE_Lock	Türschloss aufschliessen
	KEY_STATE_Unlock	Türschloss zuschliessen
VERDECK	VERDECK_Opened	Cabriolet-Verdeck geöffnet
	VERDECK_Opening	Cabriolet-Verdeck beim Öffnen
	VERDECK_Closing	Cabriolet-Verdeck beim Schliessen
	VERDECK_Closed	Cabriolet-Verdeck geschlossen

WIN_CAN	WIN_CAN_Invalid	ohne Wirkung
	WIN_CAN_Move	Befehl vom CAN-Bus zum Bewegen des Fensters
	WIN_CAN_MoveEnd	Befehl vom CAN-Bus zum Bewegen des Fensters bis zum Anschlag
WIN_MOTOR	WIN_MOTOR_Neutral	ohne Wirkung
	WIN_MOTOR_Close	Befehl zum Öffnen an Fensterhebermotoren
	WIN_MOTOR_Open	Befehl zum Schliessen an Fensterhebermotoren
ZENTR_MOT	ZENTR_MOT_Neutral	kein Befehl an Front-Verriegelungsmotoren
	ZENTR_MOT_Open	Befehl zum Öffnen an Front-Verriegelungsmotoren
	ZENTR_MOT_Close	Befehl zum Schliessen an Front-Verriegelungsmotoren
ZV_SCHL	ZV_SCHL_KeepState	ohne Wirkung
	ZV_SCHL_Lock	Befehl vom CAN-Bus zum Verriegeln
	ZV_SCHL_Unlock	Befehl vom CAN-Bus zum Entriegeln

Tabelle C.1: Verwendete Datentypen in ASCET.

C.2 Nachrichten

Nachricht	**Datentyp**	**Schnittstelle**	**Schnittstellentyp**
TSG_In_BATT[1]	BATT	CAN-Eingang	periodisches Signal
DoorLock_In_ERROR_KEY	Boolean	CAN-Eingang	spontanes Signal
DoorLock_Out_ERROR_KEY	Boolean	CAN-Ausgang	spontanes Signal
TSG_In_FCODE_T0	Boolean	CAN-Eingang	periodisches Signal
TSG_In_FT_OFFEN	Boolean	S2-Eingang	kontinuierliches Sensorsignal
DoorLock_In_FT_RIEGEL	Boolean	S2-Eingang	kontinuierliches Sensorsignal
DoorLock_In_KEY_STATE	KEY_STATE	S1-Eingang	kontinuierliches Sensorsignal
DoorLock_In_KL_START	Boolean	CAN-Eingang	periodisches Signal
DoorLock_In_MOTOR_LFT	Boolean	CAN-Eingang	periodisches Signal
TSG_In_T_OFFEN	Boolean	S1-Eingang	kontinuierliches Sensorsignal
DoorLock_In_T_RIEGEL	Boolean	S1-Eingang	kontinuierliches Sensorsignal
DoorLock_In_VERDECK	VERDECK	CAN-Eingang	periodisches Signal
DoorLock_In_ZV_SCHL_A	ZV_SCHL	CAN-Eingang	periodisches Signal
DoorLock_In_ZV_SCHL_L	ZV_SCHL	CAN-Eingang	periodisches Signal
DoorLock_In_ZV_SCHL_R	ZV_SCHL	CAN-Eingang	periodisches Signal
DoorLock_Out_B_LOW_KEY	Boolean	CAN-Ausgang	spontanes Signal
DoorLock_Out_BLINK	BLINK	CAN-Ausgang	periodisches Signal
DoorLock_Out_FZENTR_MOT	FZENTR_MOT	S2-Ausgang	kontinuierliches Stellsignal
DoorLock_Out_ZENTR_MOT	ZENTR_MOT	S1-Ausgang	kontinuierliches Stellsignal
DoorLock_Out_ZV_SCHL_L	ZV_SCHL	CAN-Ausgang	periodisches Signal
DoorLock_Out_ZV_SCHL_R	ZV_SCHL	CAN-Ausgang	periodisches Signal
DoorState_In_DOOR_STATE_VL	Boolean	CAN-Eingang	periodisches Signal
DoorState_In_DOOR_STATE_VR	Boolean	CAN-Eingang	periodisches Signal

[1]Signale mit dem Präfix „TSG“ werden gleichzeitig auf mehrere Module verteilt; in ASCET geschieht die Verteilung über Namensauflösung

DoorState_In_DOOR_STATE_HL	Boolean	CAN-Eingang	periodisches Signal
DoorState_In_DOOR_STATE_HR	Boolean	CAN-Eingang	periodisches Signal
DoorState_In_F_T_OFFEN_V	Boolean	CAN-Eingang	periodisches Signal
DoorState_In_F_T_OFFEN_H	Boolean	CAN-Eingang	periodisches Signal
TSG_In_FCODE_T1	Boolean	CAN-Eingang	periodisches Signal
TSG_In_LL	Boolean	CAN-Eingang	periodisches Signal
TSG_In_RL	Boolean	CAN-Eingang	periodisches Signal
DoorLight_Out_T_LICHT	Boolean	S1-Ausgang	kontinuierliches Stellsignal
DoorLight_Out_FT_LICHT	Boolean	S2-Ausgang	kontinuierliches Stellsignal
InteriorLight_In_KL	Boolean	CAN-Eingang	periodisches Signal
InteriorLight_Out_I_LICHT	Boolean	CAN-Ausgang	periodisches Signal
Windows_Back_In_FHB_VL	FHB	S1-Eingang	kontinuierliches Sensorsignal
Windows_Back_In_FHB_VR	FHB	S1-Eingang	kontinuierliches Sensorsignal
Windows_Back_In_FHB_HL	FHB	S1-Eingang	kontinuierliches Sensorsignal
Windows_Back_In_FHB_HR	FHB	S1-Eingang	kontinuierliches Sensorsignal
Windows_BackMotor_In_FF_BEWEG	Boolean	S2-Eingang	kontinuierliches Sensorsignal
Windows_BackMotor_In_FF_OBEN	Boolean	S2-Eingang	kontinuierliches Sensorsignal
Windows_BackMotor_In_FF_UNTEN	Boolean	S2-Eingang	kontinuierliches Sensorsignal
Windows_BackMotor_Out_FF_MOTOR	WIN_MOTOR	S2-Ausgang	kontinuierliches Stellsignal
Windows_CAN_Back_Out_WIN_HL_CL	WIN_CAN	CAN-Ausgang	periodisches Signal
Windows_CAN_Back_Out_WIN_HL_OP	WIN_CAN	CAN-Ausgang	periodisches Signal
Windows_CAN_Back_Out_WIN_HR_CL	WIN_CAN	CAN-Ausgang	periodisches Signal
Windows_CAN_Back_Out_WIN_HR_OP	WIN_CAN	CAN-Ausgang	periodisches Signal
Windows_CAN_Front_Out_WIN_VL_CL	WIN_CAN	CAN-Ausgang	periodisches Signal
Windows_CAN_Front_Out_WIN_VL_OP	WIN_CAN	CAN-Ausgang	periodisches Signal
Windows_CAN_Front_Out_WIN_VR_CL	WIN_CAN	CAN-Ausgang	periodisches Signal
Windows_CAN_Front_Out_WIN_VR_OP	WIN_CAN	CAN-Ausgang	periodisches Signal
Windows_Front_In_F_BEWEG	Boolean	S1-Eingang	kontinuierliches Sensorsignal
Windows_Front_In_F_OBEN	Boolean	S1-Eingang	kontinuierliches Sensorsignal
Windows_Front_In_F_UNTEN	Boolean	S1-Eingang	kontinuierliches Sensorsignal
Windows_Front_Out_F_MOTOR	Boolean	S1-Ausgang	kontinuierliches Stellsignal
Windows_Out_B_LOW_WIN	Boolean	CAN-Ausgang	spontanes Signal
Windows_Out_ERROR_WIN	Boolean	CAN-Ausgang	spontanes Signal
Windows_Request_Back_In_FFHB_HL	FHB	S2-Eingang	kontinuierliches Sensorsignal
Windows_Request_Back_In_FFHB_HR	FHB	S2-Eingang	kontinuierliches Sensorsignal
Windows_Request_Back_In_WIN_HL_CL	WIN_CAN	CAN-Eingang	periodisches Signal
Windows_Request_Back_In_WIN_HL_OP	WIN_CAN	CAN-Eingang	periodisches Signal
Windows_Request_Back_In_WIN_HR_CL	WIN_CAN	CAN-Eingang	periodisches Signal

Windows_Request_Back_In _WIN_HR_OP	WIN_CAN	CAN-Eingang	periodisches Signal
Windows_Request_Front_In _WIN_VL_CL	WIN_CAN	CAN-Eingang	periodisches Signal
Windows_Request_Front_In _WIN_VL_OP	WIN_CAN	CAN-Eingang	periodisches Signal
Windows_Request_Front_In _WIN_VR_CL	WIN_CAN	CAN-Eingang	periodisches Signal
Windows_Request_Front_In _WIN_VR_OP	WIN_CAN	CAN-Eingang	periodisches Signal

Tabelle C.2: Bedeutung der Nachrichten im ASCET-Modell auf Modulebene.

Abbildungsverzeichnis

2.1 Phasen der Software-Entwicklung. 12
2.2 Verteilung des Arbeitsaufwandes. 15
2.3 Das Steuergerät als Teil eines Regelkreises. 16
2.4 Kernprozess der Steuergeräte-Entwicklung. 18
2.5 AUTOSAR Software-Architektur. 20
2.6 Zuordnung von Prozessen zu Tasks. 21

3.1 Überblick über gängige Testmethoden. 25
3.2 Beispiel für einen Kontrollflussgraphen. 28
3.3 Einordnung der Testsysteme. 36
3.4 Diversifizierender Testansatz. 37

4.1 Inklusionseigenschaft von Spezifikationssprachen. 42

5.1 Funktionen des Türsteuergerätes. 46
5.2 Position der Türsteuergeräte und Bedienelemente. 47

6.1 ASCET-Toolkette. 53
6.2 Projektstruktur. 54
6.3 Definition von Prozessen. 55
6.4 Scheduling in ASCET. 55
6.5 CTE/XL Screenshot. 58
6.6 Beispiel für ein SSD. 59
6.7 Beispiel für ein STD. 60
6.8 Modelchecking in AutoFocus. 62
6.9 SSD zur Demonstation der Modelchecking-Anbindung . . 63
6.10 STD zur Demonstration der Modelchecking-Anbindung . 63

7.1 DoorLock_Module in ASCET. 70
7.2 Windows_Module in ASCET. 72

7.3 FrontMotor in ASCET. 75
7.4 DispatchCommand in ASCET. 76
7.5 Testen mit ASCET. 79
7.6 Rapid-Prototyping-Aufbau. 81
7.7 Hallsensoren zur Einklemmschutzerkennung. 82
7.8 Signalverläufe der Sensoren. 82
7.9 Zustandsbasiertes Quadrature Encoding. 83

8.1 Überblick über das AutoFocus-Modell. 86
8.2 DoorLock-Komponente in AutoFocus. 87
8.3 Windows-Komponente in AutoFocus. 89
8.4 FrontMotor in AutoFocus. 91
8.5 DispatchCommand in AutoFocus. 92
8.6 Modellbasiertes Testen mit AutoFocus. 94
8.7 Modularer Verifikationsansatz. 100

9.1 Beispielautomat zur Demonstration der Codeerzeugung. . 108
9.2 Code-Einbindung in ASCET. 109

Tabellenverzeichnis

2.1 Zuordnung von Funktionen zu Subsystemen. 18
2.2 Unterstützungsprozesse. 19

3.1 Einfache Bedingungsüberdeckung 29
3.2 Mehrfachbedingungsüberdeckung 30
3.3 Minimale Mehrfachbedingungsüberdeckung 30
3.4 MC/DC 31
3.5 Klassifikation der Testsysteme. 36

7.1 Schnittstellenkonvention in ASCET. 66

8.1 Schnittstellenkonvention in AutoFocus. 85

9.1 Zeitlicher Modellierungsaufwand. 107
9.2 Anzahl gefundener Fehler. 111
9.3 Gefundene Fehler je Zeitaufwand. 114

10.1 Testen vs. Modelchecking. 117

C.1 Verwendete Datentypen in ASCET. 127
C.2 Bedeutung der Nachrichten im ASCET-Modell auf Modulebene. 129

Literaturverzeichnis

[Auta] AutoFOCUS 2 Website. http://www4.in.tum.de/ af2.

[AUTb] AUTOSAR Website. http://www.autosar.org.

[Bal00] H. Balzert. *Lehrbuch der Software-Technik - Software-Entwicklung*. Spektrum Akademischer Verlag, 2000.

[BHS99] M. Broy, F. Huber, and B. Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, (13(3)):121–134, 1999.

[BS01] M. Broy and K. Stolen. *Specification and Development of Interactive Systems*. Springer, 2001.

[Cad] Cadence SMV. http://embedded.eecs.berkeley.edu/Alumni/kenmcmil/smv.

[CGA00] E. M. Clarke, O. Grumberg, and Peled D. A. *Model Checking*. MIT Press, 2000.

[Com] International Electrotechnical Commission. http://www.iec.ch.

[Est] Esterel Technologies. http://www.esterel-technologies.com.

[FTWT+05] S. Fürst, H. Treseler, T. Woeniger-Trausenecker, A. Buchwieser, H. Putzer, and A. Vollerthun. BMW Group Standard 95014, Embedded Software Entwicklung, 2005.

[Gä06] M. Gärtner. Efficient Development of Embedded Automotive Software with IEC 61508 Objectives using SCADE Drive. http://www.esterel-technologies.com/files/EW-2006-

Efficient-Development-of-Embedded-Automotive-Software-with-IEC-61508-Objectives-using-SCADE-Drive.pdf, 2006.

[Gmb] ETAS GmbH. http://www.etas.de.

[Gmb03a] ETAS GmbH. *ASCET V5.0 Benutzerhandbuch*. Stuttgart, 2003.

[Gmb03b] ETAS GmbH. *ASCET V5.0 Referenzhandbuch*. Stuttgart, 2003.

[Gmb03c] ETAS GmbH. *ASCET V5.0 Schnelleinstieg*. Stuttgart, 2003.

[Har01] N. Hartmann. *Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik*. PhD thesis, Universität Karlsruhe, 2001.

[Hua04] M. Huang. BMW Group, ASCET-SD 4.2.4 Modellierungsrichtlinien, 2004.

[JW01] Jan Jürjens and Guido Wimmel. Security Modelling for Electronic Commerce: The Common Electronic Purse Specifications. In Beat Schmid, Katarina Stanoevska-Slabeva, and Volker Tschammer, editors, *Towards the E-Society. Proceedings of 1st IFIP International Conference on E-Commerce, E-Business and E-Government*, pages 489 – 506. Kluwer Academic Publishers, 2001.

[Mei05] R. Meinlschmidt. BMW Group, Modellierungsrichtlinien MATLAB/Simulink/Stateflow and TargetLink, 2005.

[MIS98] MISRA. Guidelines for the Use of the C Language in Vehicle Based Software, 1998.

[MSF05] H. Mann, H. Schiffelgen, and R. Froriep. *Einführung in die Regelungstechnik*. Hanser, 2005.

[Mye01] G. J. Myers. *Methodisches Testen von Programmen*. Oldenbourg, 2001.

[PH02] B. Paech and F. Houdek. Das Türsteuergerät - Eine Beispielspezifikation. Technischer Report IESE-Report Nr. 002.02/D, 2002. in German.

[Pol] Polyspace Technologies. http://www.polyspace.com.

[RCT] RCTA, Inc. http://www.rtca.org.

[RJdlCV04] B. Remp, J. Jany, M. de la Cruz, and A. Vollerthun. Leitfaden zur Testfall-Erstellung. Technical report, BMW Group, 2004.

[SPHP02] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-Based Development of Embedded Systems. In *Advances in Object-Oriented Information Systems, Lecture Notes in Computer Science*, volume 2426, pages 298 – 311. Springer Verlag, 2002.

[SPW04] B. Schätz, M. Pister, and A. Wisspeintner. Anforderungsanalyse in der modellbasierten Entwicklung am Beispiel von AutoFocus. *Softwaretechnik-Trends 24(1)*, 2004.

[sys] systematic-testing.com. http://www.systematic-testing.com.

[SZ04] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. Vieweg, 2004.

[Tel] Telelogic. http://www.telelogic.com.

[The] The Mathworks, Inc. http://www.mathworks.com.

[UPP] UPPAAL. http://www.uppaal.com/.

[Val] Validas AG. http://www.validas.de.

[WD05] M. Wolff and H. Deiss. Neue Testmethodik für vernetzte Systeme. *Hanser Automotive*, (7), 2005.

[Wim00] Guido Wimmel. Specification Based Determination of Test Sequences in Embedded Systems. Master's thesis, Technische Universität München, 2000.

[Wim05] G. O. Wimmel. *Model-Based Development of Security-Critical Systems*. PhD thesis, Technische Universität München, 2005.